AFIT/GCS/ENG/99M-01

AN INTERACTIVE TOOL FOR REFINING SOFTWARE SPECIFICATIONS FROM A FORMAL DOMAIN MODEL

THESIS

Gary L. Anderson, B.S.
Captain, USAF

AFIT/GCS/ENG/99M-01

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government

AFIT/GCS/ENG/99M-01

AN INTERACTIVE TOOL FOR REFINING SOFTWARE SPECIFICATIONS FROM A FORMAL DOMAIN MODEL

THESIS

Presented to the Faculty of the Graduate School of Engineering

Of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Gary L. Anderson, B.S.

Captain, USAF

March 1999

AFIT/GCS/ENG/99M-01

AN INTERACTIVE TOOL FOR REFINING SOFTWARE SPECIFICATIONS FROM A FORMAL DOMAIN MODEL

Gary L. Anderson, B.S.
Captain, USAF

Approved:

Thomas C. Hartrum Chairman	5 March 1999 date
Robert F. Graham Member	5 March 1999 date
Scott A. Kuhn Member	5 Mar 99 date

Acknowledgments

I want to thank my thesis advisor, Dr. Hartrum, for guiding me through the long and challenging thesis process. He always made himself available and was pleasantly cordial but always professional. I also thank my committee members, Major DeLoach and Major Graham, who often gave me timely advice when I was "stuck".

I owe a debt of gratitude to my lovely wife, Rajni, and two children, Sheena and Kevin, who are the greatest blessings of my life. They were always encouraging and understanding through many late nights and working weekends. Most of all I want to thank the Lord Jesus. His strength helped me focus on the task at hand, while maintaining a balance and keeping a healthy perspective on life.

Table of Contents

List of Figures

Page

Abstract

This work examines the process for refining a software specification from a formal object-oriented domain model. This process was implemented with interactive software to demonstrate the feasibility and benefits of automating what has been a tedious and often error-prone manual task.

The refinement process operates within the framework of a larger Knowledge-Based Software Engineering system. A generic object-oriented representation is used to store a domain model, which allows the specification tool to access, select, and manipulate the required objects to form a customized specification. The specification is also stored as an object-oriented model, which in turn can be accessed by a design tool to transform the specification into source code.

The tool has been designed as an interactive program that helps guide the user through the process of building the specification. The tool has been named the Elicitor-Harvester because of the functions it performs. It elicits application requirements from the user and harvests pre-existing knowledge from the formal domain.

An Interactive Tool for Refining Software Specifications from a Formal Domain Model

1 Introduction

This work examines a process for specifying software applications. An automated tool was built to interactively guide a user through the process of refining software specifications from a formal object-oriented domain model. The tool allows the user to choose the parts of the domain model needed for the application, modify those parts as needed, and define new components to supplement the specification

1.1 Background

The article "No Silver Bullet", written in 1986, explained the difficulty the software industry was having trying to keep up with the incredible performance increases the computer hardware industry has achieved [19]. While the hardware industry continues to exploit new technologies and improve manufacturing techniques, the software industry continues to grasp at many new techniques and methodologies, hoping to find the "Silver Bullet", so to speak, that can kill the monster that plagues software development. The article explains that all these efforts have focused on the accidental difficulties of software (those problems associated with building the code) and not on the essential difficulties (inherent in the nature of the software). These essential difficulties lie in the complexity, conformity, changeability, and invisibility of software. The hardest part of software development lies in understanding, identifying, and specifying the requirements and rules of the desired software solution, and in making upgrades and changes once the product is fielded [2]. Knowledge Based Software Engineering (KBSE) is an attempt to address these essential difficulties.

KBSE is the study of representing information gained from domain knowledge and a problem statement with a series of formal models in an attempt to apply automated manipulation to the software development process. The formal representations can also allow Artificial Intelligence (AI) reasoning techniques and formal methods to be applied to determine different levels of correctness and completeness of the specification, design, and implementation [1]. By automating the transformations from formal specification through design and implementation and into source code, software systems can be maintained at the formal specification level instead of at the code level as it is currently. By maintaining provably correct

transformations from specification to code, the verification of source code is implicit and the need for verification testing (building the system right) is eliminated. Validation testing (building the right system) then becomes the main form of testing and would be an exercise of iteratively adjusting the specifications to meet the end users' requirements. Each time the specification is changed, the automatic transformations rebuild the source code to match the specification.

The goals of the Air Force Institute of Technology (AFIT) KBSE research are to address some of the essential difficulties of software development by applying formal methods and automating the transformations as much as possible to avoid problems as described by Grassmann and Tremblay.

> During development of information systems, many problems arise from inadequacies of the notations that are used to describe the software product at each stage of the development life cycle. Many of these notations include natural language as a vehicle for describing the different artifacts. The notations that are most dependent on natural language are those that are used upstream in the life cycle (i.e., in its early phases of development). Also, in many development approaches to producing software, a different notation is used for each phase of the life cycle, and because of the very "visible seams" between phases, interface errors usually result [15].

The AFIT KBSE model consists of a series of stages beginning at the acquisition of knowledge from a Domain Expert (probably the end user of proposed software system) and ending with executable code on the fielded computers. In each stage, the knowledge data is transformed by a process that refines the knowledge representation a step closer to the final goal of an executable system. The entire process is known as the AFIT Forward Engineering Concept and is shown in Figure 1. Previous research at AFIT has demonstrated the ability to build formal domain models with a formal specification language such as Z (pronounced zed) or an algebraic language such as Larch or Slang, and parse them into a tree structure called an Abstract Syntax Tree (AST) [16] [17] [18]. Once a domain is represented in an AST, it can be accessed and manipulated in software. The square boxes in Figure 1 represent transformation processes that are candidates for automation.

As more emphasis is placed on the correctness of the specification, the transformation shown as "Problem Setting" in Figure 1 becomes very critical. The Formal Specification model represents a complete and concise description of the objects, methods, states, and events that will exist in the final system and describes *what* they should do. The problem setting stage is performed by an application engineer who

manipulates the domain model knowledge based on a set of requirements found in the Problem Statement. The Formal Specification specifies a customized application and will usually consist of a subset of the domain model objects with several details defined for the specific application. Once the Formal Specification is finalized, it passes into the Design Phase where the formal specification is transformed into a design specification.

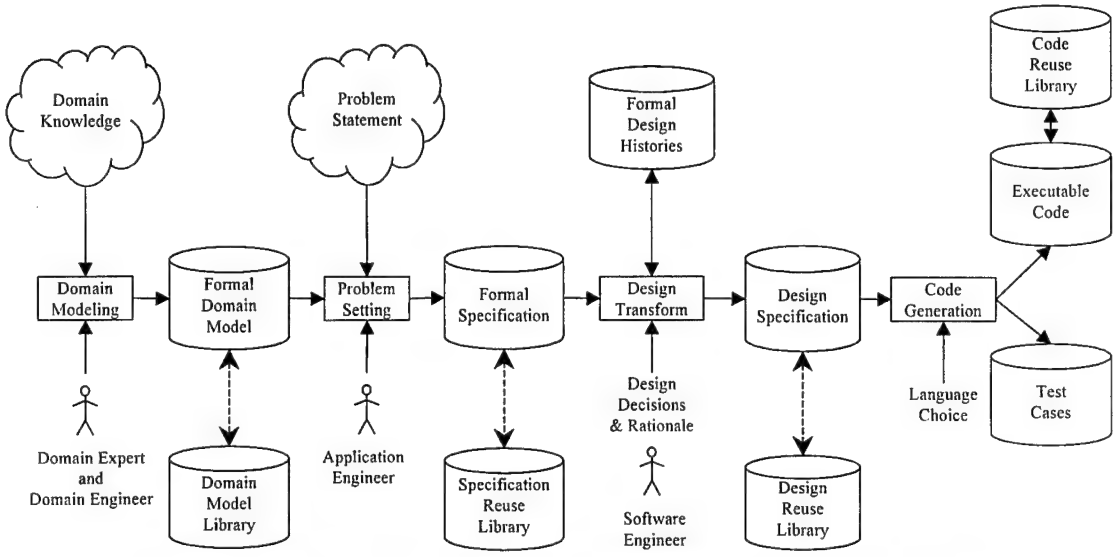

Figure 1 Formal Approach to Creation of Correct Domain-Specific Software

As the need for representing a software specification becomes critical, the need to automate the specification process increases. An application engineer can quickly become overwhelmed with the thousands of specification details if the process is performed manually. A tool is needed to guide the engineer through the process of identifying and refining specification components from the domain model. This research focuses on defining the requirements and demonstrating the feasibility of a tool to automate the Problem Setting Transformation process. Since the function of this tool is to elicit requirements from the application engineer based on the problem statement, and harvest information and knowledge contained in the existing domain model, the tool is called an Elicitor-Harvester (EH).

1.2 Problem

Elicitor-Harvester type tools have been developed for very restricted, well-defined applications such as the XCON system developed by Digital Equipment Corporation to assist in the configuration of newly ordered VAX computer systems [5]. The problem lies in trying to generalize the tool enough so it will work on a

General Object Model to be used as a knowledge store in a knowledge-based system like the one under development at AFIT. The large number of rules needed for such a general system could cause an AI engine to be overwhelmed. The question of how to represent the rules in the object model in such a way that an AI search engine can correctly identify the many possible areas that need to be specified can be quite complex. The hardest problem may be trying to determine how the tool should interact with the application engineer to identify the parts of the domain model needed to implement the desired specification. If the application engineer is not a domain expert, the EH needs to shield him or her from the confusing details about the underlying domain AST. The EH needs to prompt for information in such a way that the user does not get frustrated with the process. This research analyzes these problems and attempts to find some feasible solutions.

Problem Statement:

Demonstrate the feasibility of an Elicitor-Harvester tool using AI techniques to allow a user to create a formal specification from a well-defined domain tree. The user may not know the details of the domain model, so the EH must guide him or her through the process by prompting the user for necessary inputs. However, EH must be able to harvest pertinent knowledge from the domain tree so as to avoid burdening the user with too many questions.

1.3 Initial Assessment of Past Effort

Elicitor-Harvester has been the thesis topic of three AFIT Master's theses since 1995. Charles Wright performed an analysis of EH in 1994-1995 before the AST structures were developed. He described the EH as a tool to help build a software system by reusing existing components. The EH would elicit requirements from the user and use AI techniques such as forward chaining and search methods to automatically select reusable components that would meet the user requirements [3].

Jerry Cochran studied how an EH tool could be used to reuse object oriented components by applying some rules and predicate logic. He created an object model of a pump system to use as an example [6]. His research showed some positive results, but the capabilities of an EH need to be generalized to be able to work on different domain models.

Timothy Karagias finished his follow-on research in December, 1996 [2]. He studied ways to apply EH techniques to an object-oriented AST, which stored domain knowledge. His major goals were to define the

requirements for the EH, design a nominal system, and demonstrate the feasibility for implementing the EH. He concluded that his goals were mostly met, but more work needed to be done in several areas.

- The EH needs better verification of new aggregates created during the specification process.
- A user interface should be built to help a user along in the complicated specification process.
- The EH should be extended to make it compatible with several types of formal languages.
- He also concluded that incorporation of AI is needed to select components of the domain AST that need further specification from the user.

These three previous efforts demonstrated the feasibility and highlighted the usefulness of an EH tool. However, there were many difficult and untested problems associated with a smart interactive tool. An approach was needed that could lead the user through the entire process of identifying the input, output, and internal operations of the system; selecting the domain items that support those operations; modifying the existing operations or creating new ones to support the particular specification; then saving either the specification in work or the selected objects of the final specification. The operations are probably the most complex part of the domain model because of the formal predicates stored as pre-conditions and post-conditions. These predicates can be stored in a wide variety of formats and contain variables that can represent a variety of domain items such as input or output parameters, class attributes, classes, constants, or associations – all of which are associated with a data type or class. Since operations are so complex and important to the specification, an approach for handling operations in the EH process needed to be studied.

1.4 Proposed Solution

Without a tool to automate the specification process the engineer must use a manual process that is slow, tedious, and error prone. The manual process would require the engineer to find and modify domain description text files, check several separate files for consistency, compile the files and parse the specification descriptions through a domain parser.

This research proposes to define an Elicitor-Harvester process to access domain knowledge acquired during the requirements gathering phase, which is represented as an AST. This knowledge is then transformed into a formal specification AST by eliciting the problem statement decisions from the application engineer and harvesting knowledge from the existing domain. Through a series of questions an EH allows the application

engineer to select and refine the existing domain components into a formal specification that represents the final software product.

1.5 Scope

A fully operational EH would need to have the capability handle all types of domain information associated with object-oriented modeling. This information includes aggregate and primitive classes, inheritance between sub and super-classes, associations between classes and their multiplicities, associative classes, class attributes, class constraints, operations or methods with their pre and post conditions, parameters, private and global constants, private and global data types, states, events, and state transitions. An EH would also need a well-designed user interface with the associated error checking to interact with the user. The EH would also need to perform many checks for consistency among the specification objects and perform various initialization and cleanup functions. The task of implementing all functions needed for an EH was much too large for this thesis effort.

The goal, therefore, was to thoroughly define the requirements of a general EH tool, while choosing a challenging subset of the proposed functionality that could demonstrate successful design and implementation of the EH concept. Operations and data types were the two parts of the domain model that were chosen for demonstration purposes, since their representations are fairly complex and they can be defined in a large variety of ways. Operations contain pre-conditions and post-conditions defined as formal predicates in the domain model. These predicates and data types relate to many other parts of the domain model, which made them quite complicated to implement, but very important in the specification process. A long-term goal for a user interface would allow the user to input predicates as natural language descriptions and translate them into formal language definitions with an expert system. This research had to limit the user to input predicates in proper Z[1] specification notation that could be correctly parsed in by an existing Z parser.

This research effort also focused on incorporating Artificial Intelligence methods into the Elicitor-Harvester, mainly to deal with the rules used for adding or modifying operations and data types. Since the EH

[1] Z (Pronounce zed) is a formal specification language initiated by Jean-Raymond Abrial and subsequently developed by a team at Oxford University. It is based on logic, sets, relations, and functions and is used for stating what a system should do and in what order it should be done without stating how it should be done. [15]

needs a well-defined domain as an input, two existing domains that were already analyzed and defined at AFIT were used for demonstration purposes – the School and the Cruise Missile domains. These AFIT defined domains are represented as Z schemas in LaTeX[2] files and can be parsed into a Domain Object Model (DOM) AST. While developing AI techniques for the EH on these specific test domains, operations that could be generalized to any type of domain vs. methods that should be restricted to a specific domain were identified.

1.6 Approach

To meet the proposed research objectives the following approach was followed:

1. *Became familiar with the current KBSE domain models and tools* – Studied the current AFIT literature available for the current KBSE resources. Became familiar with the structure of the domain AST and the code that generates it. Also became familiar with analysis tools.

2. *Studied previous EH research and analyzed existing EH software available at AFIT* – Read the three previous thesis efforts performed by AFIT graduate students to gain an understanding of areas needing further research. Became familiar with the existing EH tool created by Karagias and Hartrum to identify and narrow the scope of study.

3. *Studied other research performed outside AFIT* – Performed a literature search for other research in the area of component reusability in general and EH techniques in particular.

4. *Studied Artificial Intelligence techniques* – Performed a literature search for AI tools and techniques that could be integrated into an object-oriented EH tool. Sought guidance from committee members concerning possible AI options.

5. *Defined requirements for an EH tool* – Several tasks needed to be performed to complete the requirements definitions as listed below.

    - Defined the specific goals this research tried to demonstrate.
    - Documented requirements that identified the scope and expectations for the EH.
    - Specified some test cases to validate the results.
    - Identified formal domain models to be used to test the EH.

[2] LATEX is a special version of the TEX, which is a trademark of the American Mathematical Society.

- Defined the structure of the domain AST to be used as input to the EH and the specification AST to be output from the EH.

6. *Designed and Implemented the EH tool* – Used knowledge acquired from the previous steps to choose the tools and languages to facilitate development of an EH. Then proceeded to design and code a tool that met the requirements defined during step five.

7. *Tested EH software on domain models* – Used the test cases identified during step five to validate the EH operations. Most testing was informal debugging and was an ongoing activity throughout the design and implementation phase. Requested other students and committee members to use the tools and provide feedback about the ease and understandability of the EH tool.

8. *Analyzed results* – Compared findings with the goals defined in step five and described how well the results satisfied those goals. Analyzed whether or not the requirements were met or how close they were to being met.

1.7 Assumptions

Well-defined object oriented domains were assumed to exist and to be available at AFIT. It was assumed that the domain models used for testing the EH included aggregation. The domain AST was expected to sufficiently support all types of domain model objects.

1.8 Thesis Overview

Chapter 2 describes the operating environment of the EH providing the reader with background information necessary to understand the setting of the problem. The languages and data structures used, other KBSE tools integrated, rule-based techniques used, and other relevant research are discussed. Chapter 3 discusses the functionality required for an EH and the philosophy that guided the decisions made about the requirements. Chapter 4 describes the design of the EH and the reasons for the design choices. Chapter 5 identifies the functional capabilities actually implemented to demonstrate feasibility and several of the problems encountered during implementation. The testing and evaluation of the EH is also discussed in Chapter 5. Finally, Chapter 6 closes with remarks about the accomplishments of this work and recommends further study in several areas relating to an EH.

2 Background

The purpose of the EH tool is to allow application engineers to develop their own customized software to satisfy their particular needs. The EH should use the formal domain representation to extract parameters the application engineer may need to identify for his or her particular system. The EH should interactively prompt the application engineer to specify certain parameters for the system being created based on an end user's requirements. AI techniques should be used to aid the application engineer in developing formal specifications from a domain model and customer requirements. The EH should search the domain AST for incomplete specifications or places where choices must be made. A user interface (preferably graphical) should guide the user through the problem areas that require decisions. As the user inputs specifications, the EH should mark portions of the domain model that need to be added to the specification model, helping it to evolve into a formal specification AST [4].

For example, AFIT may decide to build an academic domain model as a baseline to develop customized systems for various departments within AFIT. An object model would be built to describe the important entities and relationships in AFIT. Once this domain model is completed, application engineers could use the EH to identify the subset of objects, relationships, and operations they need for their particular software sub-system. Let's say a user needs a tool to track student grades. The student's GPA could be used to identify whether the student is in the top 10% or needs to be put on probation. During the process, the sub-system designer would be prompted to enter specifications such as GPA threshold for probationary status. The EH would build a formal specification AST, which could then be used to automatically create the executable code for the grade tracking system.

In order to perform such intelligent functions, AI techniques must be incorporated into the EH to allow it to search the AST for areas that need to be specified. The EH should help an application engineer automatically find parts of the domain that were intentionally left unspecified because of the need for flexibility in the specification phase. For instance, a knowledge-based system used to specify a computer system would not know the specific power supply or hard drive required until a user is ready to design the system. The EH should be able to find these instances of unspecified information and prompt the user for input. The EH should iteratively add the knowledge provided by the user to the AST during this specification process. The EH should also allow the user to query the knowledge base to check the current specifications.

Once the formal specifications are complete, the KBSE system automatically transforms the specifications into a formal design [1]. From there, an automatic code generator creates the executable code. The knowledge based tool suite should also provide a means to test the final code against the requirements for validation purposes.

This chapter describes research directly related to the Elicitor-Harvester. The key pieces that come together to implement the EH include Knowledge Base Data Dictionary, user interface, rule-based inference engines, rule bases, domain model, Abstract Syntax Trees (ASTs), REFINE™[3] (the language used to implement the EH), Common Lisp©[4] (for string manipulation and file I/O), the KBSE environment and supporting software (parse tools, grammar checker, POB save, domain definitions, Z notation), history maintenance, object reuse, and object (knowledge) transformations.

Information directly related to EH is almost exclusively generated by AFIT research, which includes three previous Master's theses, published papers, and several formal and informal papers and drafts. Articles in the area of software reuse and AI were considered if they discussed the elicitation of domain knowledge or knowledge based AI techniques.

2.1 Languages Used

To demonstrate the effectiveness of an automated EH process, it was necessary to use a computer language that would provide the ability to manipulate information stored in the domain model. The majority of the software currently supporting the AFIT KBSE program is written in the REFINE language. REFINE provides many functions to store, create, and manipulate tree structures, which gives users great flexibility to define the meta-model structures for formal specifications. REFINE is written in Common Lisp, which has an extensive built-in library of functions that can be utilized in REFINE programs when needed.

2.1.1 REFINE

REFINE is a programming environment, which provides a programming language and a set of language processing tools (parser, compiler, etc.). The REFINE language is the first programming language to

[3] REFINE is the registered trademark of Reasoning Systems, Palo Alto, California.

[4] COMMON LISP is copyrighted by the Digital Equipment Corporation, 1984.

provide an integrated treatment of set theory, logic, transformation rules, pattern matching, and procedure. Because the language is executable and allows you to write programs at the specification level, the REFINE system supports programming with "executable specifications" [20].

REFINE does not support most of the data structures that computer programmers are used to such as arrays, records, or linked lists, but allows the user to define object classes and maps between objects and types, which can simulate most data structures needed. A declared object class along with all its related maps can be thought of as a record structure similar to those used in Third Generation Languages (3GLs) such as Pascal, C, or Ada. Figure 2 shows comparable record declarations for Ada and REFINE.

```
TYPE DepArray IS
  ARRAY (1..10) OF STRING

TYPE Employee is RECORD
  ID          : Integer;
  Name        : NameType;
  Gender      : CHARACTER;
  Dependents: DepArray;
  PayRate     : Float;
END RECORD;

Clerk  : Employee;
Janitor: Employee;

Clerk.ID := 1234;
```

Ada Record

```
var Employee   : object-class subtype-of user-object
var ID         : map (Employee, Integer) = {||}
var Name       : map (Employee, NameType) = {||}
var Gender     : map (Employee, symbol) = {||}
var Dependents: map (Employee, set(string))
                  computed-using Dependents(x) = {}
var PayRate    : map (Employee, Real) = {||}

var clerk   : Employee = make-object('Employee)
var janitor : Employee = make-object('Employee)

ID(clerk) <- 1234
```

Equivalent REFINE structure

Figure 2 Declaring a REFINE data structure

If *Employee* is thought of as the object class, then the maps define the attributes of the object class. All attributes of the instantiated *clerk* and *janitor* objects in Figure 2 are initially undefined, which means a value is not currently mapped to the object class. REFINE supports sets and sequences of objects as shown in the *Dependents* attribute. REFINE also allows the designer to define the object-oriented concept of inheritance by declaring an Object-Class the subtype of another object class. The subtype will inherit all attributes from its ancestors up the inheritance chain. Object classes and maps are used extensively in AFIT KBSE code. They are the building blocks for Abstract Syntax Trees (ASTs), which are used to store object models, grammars, and even the source code itself.

2.2 The Abstract Syntax Tree

Abstract Syntax Trees allow convenient and structured ways to store and represent data so it may be operated on with software. The nodes of the tree are defined by declaring a variable as an object class type and the connections between the nodes are defined by declaring a variable as a map between two nodes as shown in Figure 2. The Employee structure as declared is not an AST. If a user wants the Employee class to be an AST, the attributes must be identified in a *Define-Tree-Attributes* statement – then the Employee AST could be represented as shown in Figure 3. Notice the NameType node could also be declared as an object-class and could have tree attributes below it such as FirstName, LastName, and MiddleInit mapped to strings.

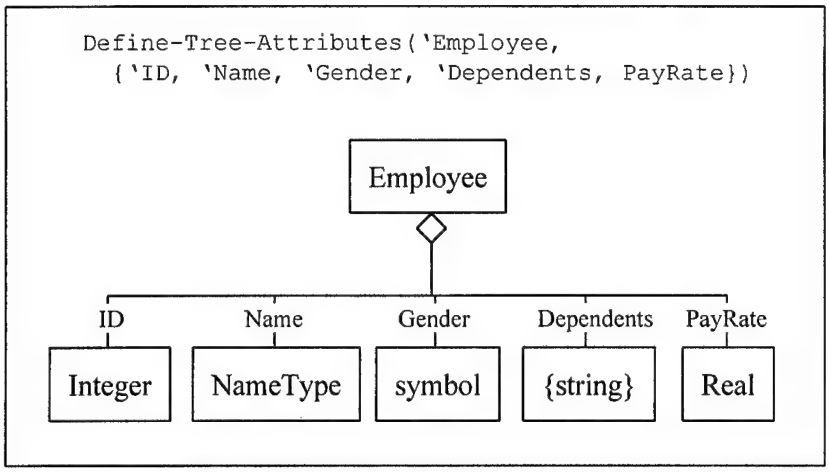

Figure 3 Employee object class defined as an Abstract Syntax Tree

When a structure has been declared as a tree, the REFINE tree traversal functions can be used on the tree to perform searches, copies, comparisons, or other operations on the nodes of the tree. Not all attributes must be declared as tree attributes in the *Define-Tree-Attributes* statement. Non-tree attributes can be accessed the same as tree attributes, but they will not be visited during the tree traversal functions. Since the AST is the main storage structure in REFINE, AFIT KBSE system employs them to represent all object models and grammars.

2.2.1 Accessing nodes in the tree

While a REFINE session is running, only one object instance can be the "current node". The current node must be some object class type defined by the user or a REFINE defined object class. Attributes of the current node can be set, changed, or deleted with assignment statements. For example, if an Employee object is

the current node, then `PayRate(current_node) <- 14.75` would set the employees pay to $14.75. Important nodes such as the root of an AST can declared as global variables with the `var` declaration shown in Figure 2 as:

```
var clerk : Employee = make-object('Employee).
```

An attribute can be a set or sequence that can have new elements added to it. E.g. if the `clerk` had a new baby, the Dependents set could be updated with this statement:

```
Dependents(clerk) <- Dependents(clerk) with "John Jr."
```

2.2.2 Saving the AST

Often when using an object base represented in an AST, it is desirable to save the object base to a permanent file. REFINE provides the functionality to save objects and their tree and non-tree attributes to a file called a Persistent Object Base (POB). The POB file can be parsed back into main memory to work on in a later REFINE session. This ability is very important when working on a large domain that can't be finished in a single session.

2.3 Z Specification Language

Z is a formal specification language based on logic, sets, relations, and functions for stating what a system should do and in what order it should be done without stating how it should be done. Z, therefore, is considered to be a declarative language, as compared to a procedural or imperative one such as Pascal. In specifying a system in Z, issues concerning efficiency and implementability of the system are not of importance at the specification stage of software development. Z has become one of the most popular specification languages in recent years [15].

Z is used extensively in the AFIT KBSE system to specify various domains used as research examples and in software engineering classes. The two domains used in testing this EH research, the cruise missile and the school domains, are specified using Z notation. The Z descriptions are typed into a LaTeX text file template then loaded into an AST using Z parser and grammar developed at AFIT [16]. The EH uses these Z tools when reading in predicates typed in by the user to describe constraints and functions of a specification.

Of course, there are other formal methods for defining software specifications such as algebraic languages like Slang and Larch, but this version of EH uses Z since it is the basis for the existing AFIT toolset, and more complete domains are available for testing and analysis.

2.4 The AFIT KBSE System Representation

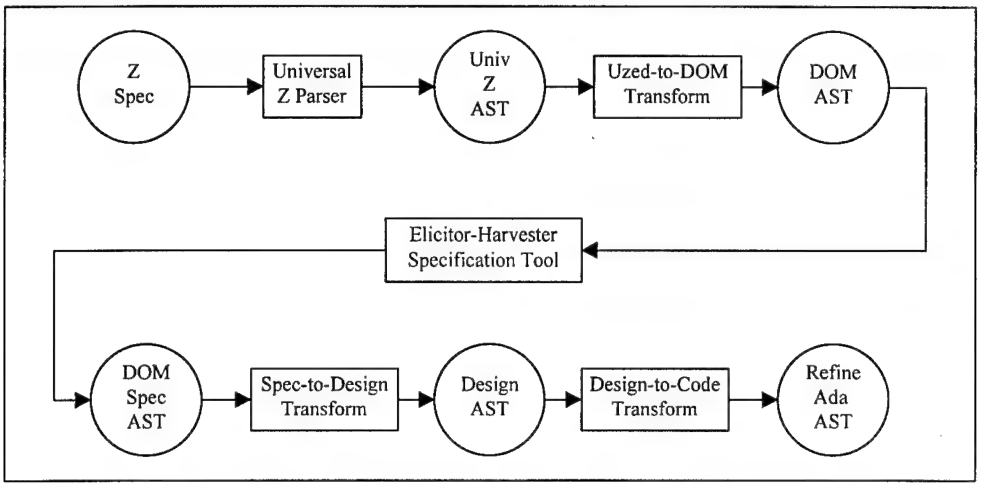

Figure 4 Transformation Process: From Formal Specification to Code

The AFIT KBSE system uses many ASTs to store information at the various development stages. Figure 4 shows the ASTs representing the development stages and the processes that transform knowledge between the ASTs. The original Z-LaTeX schemas are parsed into a Unified Object Model AST, which is transformed into a Domain Object Model (DOM) AST. The DOM is the general representation that can store any type of object-oriented domain no matter what formal specification language is used assuming the transformation software exists. The EH operates on the DOM and generally uses a subset of the DOM objects for the specification AST, although some components may be added or modified. The specification AST uses the same meta-model as the DOM. During the design stage, the specification stored in the DOM format is transformed into the Design Model, which represent a high-level design specification. During the DOM-to-Design transform, state transitions and attribute constraints are changed into functions, and Get and Set functions are created to retrieve and set the values for all class attributes.

2.4.1 Domain Object Model

The DOM, whose structure (meta-model) is shown in Figure 5, is the AST acted upon by the Elicitor-Harvester. The DOM is the generic object-oriented representation of a domain and stores all knowledge about

the given domain deemed important enough to maintain for resulting software applications. The DOM is typically populated by parsing in the Z schemas that represent the domain knowledge through the Universal Z parser and the Uzed-to-DOM transform, but can also be populated by other means using REFINE functions or constructs called forms.

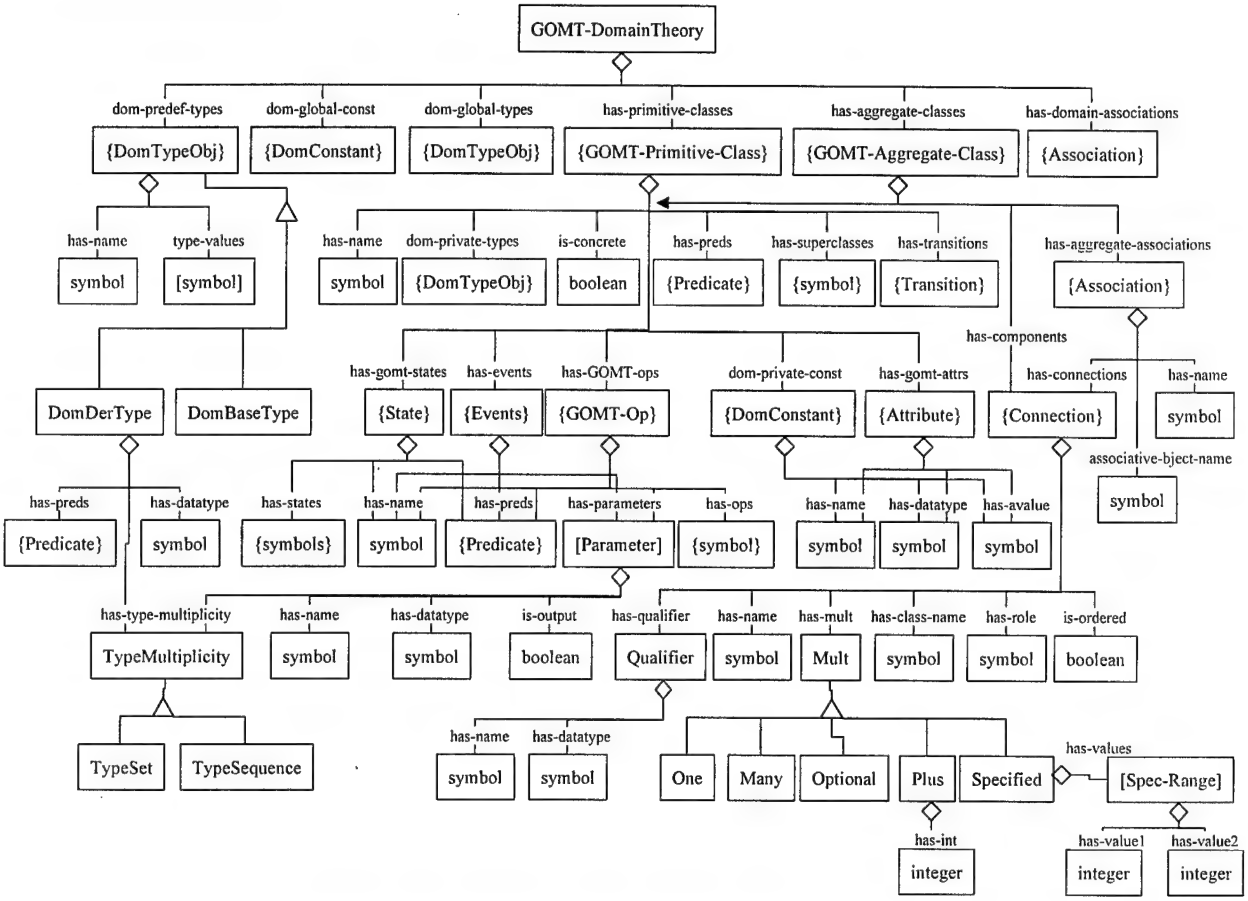

Figure 5 The Domain Object Model (DOM) AST Structure

The DOM stores global information such as classes, global types and global constants below the Domain Theory (root) level. The class is the main component of the DOM and represents real-world entities or concepts. The most significant parts of the class are the attributes and the operations that act on the attributes. Classes also have states and events, which cause transitions between states. There can be two types of classes in a domain: primitive classes and aggregate classes. An aggregate class contains attributes whose data type is another class or set of classes in the domain. A primitive class has no references to other classes. Associations represent some relationship between classes. Associations usually connect two classes and there is some

multiplicity represented in the connection; for example, a person can have zero, one, or more children. If a connection is a set, it can be ordered.

The data type is another important part of the DOM. Attributes, constants, and parameters have data types, which can be base types or derived from a base type. Type definitions can be quite flexible, which is reflected in the complexity of the *DomTypeObj* subtree. A more thorough description of the DOM can be found in *An Object Oriented Formal Transformation System for Primitive Object Classes* [1].

All object classes in the domain AST are subtypes of some parent object and thus inherit attributes. The inheritance tree hierarchy is shown in Figure 6. The user-object is the standard REFINE object class from which all user objects can inherit. All objects in the domain AST except the root node and predicates are subtypes of the Obj-Object class and thus share many inherited attributes.

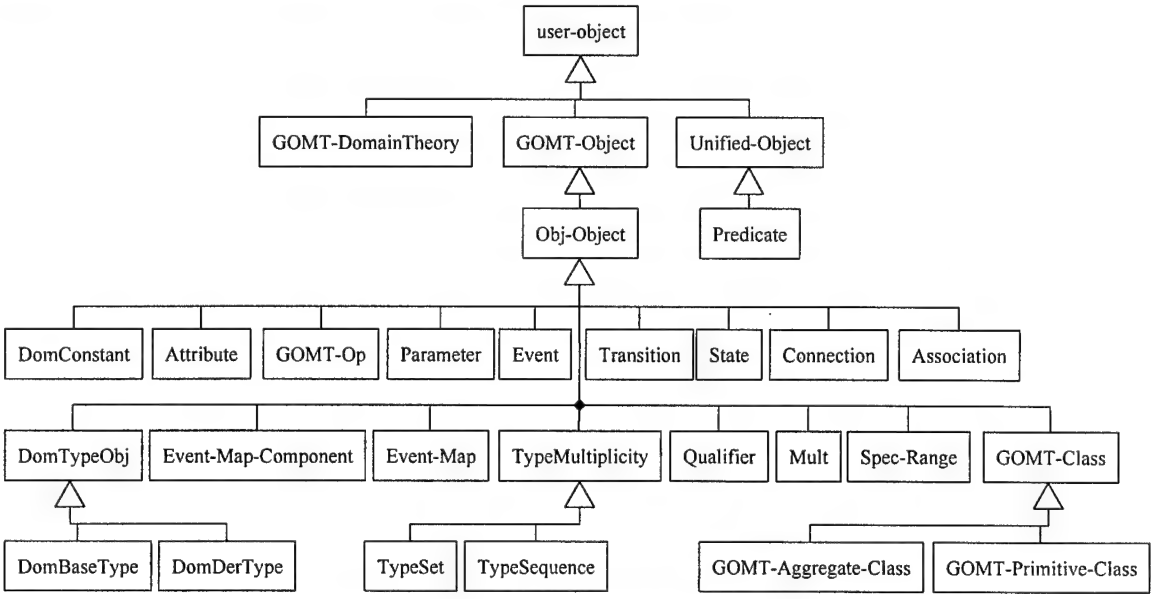

Figure 6 Domain Tree inheritance hierarchy

2.4.2 Unified-Object Model

The *Unified-Object* model is the AST structure that stores Z specifications when parsed from LaTeX text files, shown as the *Univ Z AST* in Figure 4. Predicates are subtypes of Unified-Objects, which are defined below the user-object in Figure 6.

2.4.2.1 Input Grammar/parsing

REFINE provides good support defining input and output grammars. A programmer defines a series of productions that define acceptable patterns of text read from a file or input string. The grammar for the Z parser and the Unified-Object model was created for the AFIT KBSE system by Wabiszewski [16]. Once the domain is represented in the DOM, there is generally no more use for the Z parser. However, *predicate* objects have not yet been separately defined in the DOM because they are very complex. For the sake of efficiency, predicates from the Unified-Object model are grafted directly to the DOM object that contains them.

Since predicates are used to identify such important specifications as constraints on attributes, states, events, and derived data types; pre and post conditions of operations; and guard conditions of transitions, a user of an EH would certainly need to create or change them while defining a specification. For this reason, the Z parser is employed by the EH to verify correct Z notation when an application engineer enters a new predicate.

2.4.2.2 Predicates

Z predicates are used in several contexts in the DOM. Since Z uses many special logical and mathematical characters not recognized as ASCII text and the Z parser can only read from regular text, LaTeX has defined certain text strings as substitutes for special characters. The following examples show the Z notation and the actual text strings accepted by the parser for several types of predicates.

- Consider a derived data type called *WeightType* derived from type REAL. *WeightType* should consist of only positive real numbers and zero. The Z predicate is $\forall x : WeightType \mid x \geq 0$ and the parser accepts `\forall x : WeightType | x \geq 0`.

- Consider the attribute *GPA* (Grade Point Avg.) that must be between 0 and 4.0. This invariant constraint is defined with the Z predicate $GPA \geq 0.0 \land GPA \leq 4.0$ and the parser accepts (`GPA \geq 0.0 \land GPA \leq 4.0`).

- Consider an operation called *CalcTotalTankWeight*, which calculates the output parameter *fuel_tank_weight* by multiplying the amount of fuel with the fuel density then adding the weight of the empty tank. This post condition is defined with the Z predicate:
  $fuel\_tank\_weight = fuel\_level \times fuel\_density + tank\_weight$ and the parser will accept exactly the same thing because there are no special characters.

- Consider an operation in an aggregate class whose output parameter, *ms*, is the number of Master's degree students in a set of students that is the input parameter called *advised*. *Student* and *GradClass*

are primitive classes in the domain and *member_of* is the association between them. This post condition can be defined with the Z predicate:

$$ms = \#\{s : Student; c : GradClass \mid s \in advised \wedge (s,c) \in member\_of \wedge c.program \neq DS\}$$

and the parser will accept `ms! = \# \{ s: Student; c: GradClass | ( ( s \in advised? \land ( s, c ) \in member_of ) \land c.program \neq DS ) \}.`

Figure 7 shows the AST structure of the Z predicate:

```
prop_wt! = fueltank.CalcTotalWt + jetengine.engine_weight
```

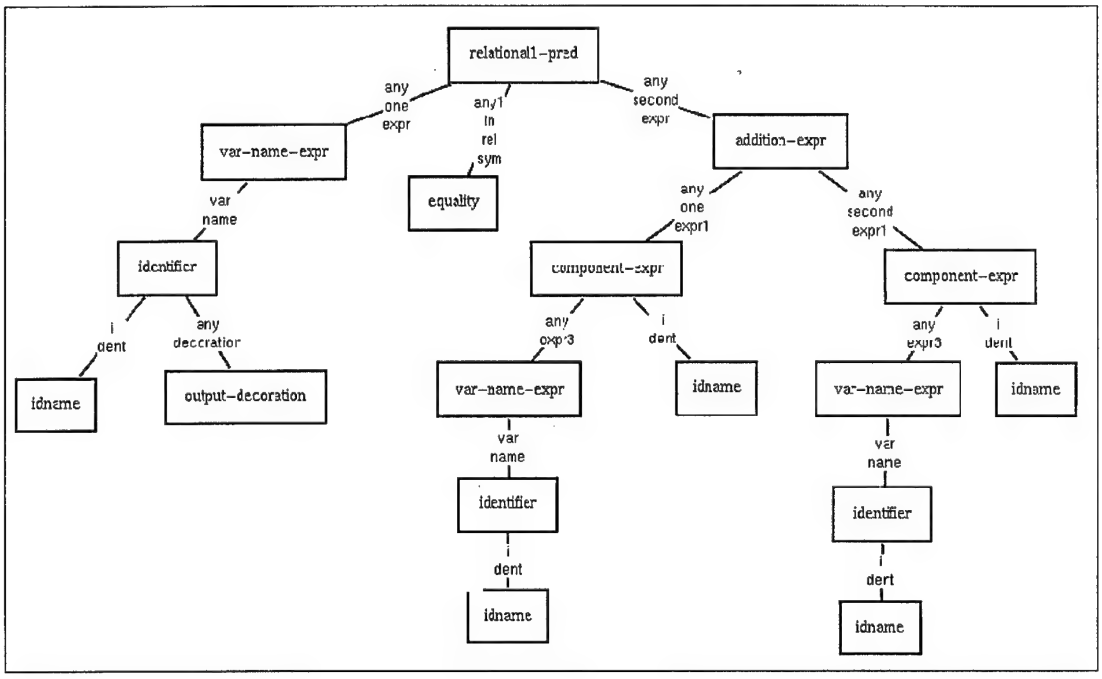

Figure 7 A Predicate AST parsed into the Unified-Object Model

It becomes obvious, after seeing a few examples, that there is a fairly steep learning curve associated with Z. An application engineer would need to be well versed in Z to specify an application using the EH with the Z parser. A better option would be to use an expert system to translate natural language specifications into Z specifications, but that tool hasn't been built yet and is outside the scope of this thesis effort.

2.5 Rule-Based Artificial Intelligence

The AFIT KBSE group has put great effort into building a general object model using REFINE ASTs to represent knowledge bases. Although much work has been done in defining the domain model

representation, parsing in Zed specifications, and AST transformations, little research has been done to apply Artificial Intelligence (AI) techniques to aid knowledge acquisition or specification refinement functions.

The Artificial Intelligence portion of a knowledge-based system consists of an inference engine and a knowledge base. Closely associated with the intelligent program is a database or fact base [7]. In a rule-based system, the knowledge base is the set of rules and associated functions that act upon the database. The inference engine is the functionality that controls how the rules are checked and performs conflict resolution to decide which rule to execute if more than one rule is satisfied. The fact base or database maintains the current status of the reasoning process. During the EH process the database is the DOM specification tree or some other temporary object created to store information while performing rule-based reasoning.

The structure of a rule is usually in the form of an implication in predicate logic, that is, *if* a set of premises in the antecedent evaluates to true *then* perform the actions given in the consequent.

REFINE rules are structured like this:

```
rule Modify-Name-Rule(X: object)
  chosen-option(X) = "Change Name" & mod-done?(X) ~= True
    --> Modify-Name(X) & mod-done?(X) = True & mod-name-done?(X) = True
```

A REFINE rule requires a name and consists of a single transform statement, which is actually a logical implication. The premises of the implication make up the antecedent and the action is the consequent, which often adds a new fact to the fact base. The parameter *(X: object)* is the object passed into the rule from the inference engine. Rules operate on the parameter object, which is generally part of a fact base or database. Since the rule can contain only a single transform statement, it is somewhat limited in the actions it can perform. However, if the rule must perform several actions in the consequent or complex checks in the premises, the rule can call functions, which can be as complex as desired.

Two different techniques are commonly used in rule-based systems: forward chaining, and backward chaining. The following sections discuss the differences between these two approaches.

2.5.1 Forward Chaining

The forward reasoning concept starts from a set of data collected through observation and works toward a conclusion. A set of rules is checked to see if the observed data satisfies the premises of any of these rules. If a rule is satisfied, it is executed to derive new facts that might then satisfy the premises of other rules to derive additional facts [7]. Since the reasoning progresses in a forward manner from the antecedent to the

consequent, and causes other rules to fire in a sort of chain reaction, the method is often called forward chaining.

The REFINE language supports forward reasoning with built-in tree traversal functions and rule constructs, but backward reasoning is not supported. REFINE has built-in tree traversal functions called *preorder-transform* and *postorder-transform*, which essentially function as the inference engine. These functions take two arguments: the first argument is an instance of some object (usually in an AST), and the second argument is a list of rule names. These *preorder-transform* and *postorder-transform* functions will traverse the AST starting at the given node in a pre-order or post-order fashion, applying the entire list of rules to each node of the tree. REFINE attempts to apply the rules of the rule list in order and will continue looping through the rule sequence until no more rules can be successfully applied. At this point the traversal function moves on to the next object in the tree and iterates through the rules again. Each time a rule successfully executes (fires), the rule consequent will perform some action, which generally consists of some adjustment to an object in the database.

2.5.2 Backward Chaining

Backward reasoning is used instead of forward reasoning in the cases where little data is known about the problem up front. The process starts with little or no data defined for attributes in the database. Instead, a goal or list of goals (or possible conclusions) to be derived by the system must be provided [7]. The backward chaining process starts off with a goal that needs to be achieved and attempts to derive that goal with the following algorithm:

1. Form a temporary stack initially composed of all top-level goals defined in the system.
2. Set the goal to be traced equal to the top goal on the stack. If the stack is empty (i.e., all top-level goals have been tried), halt and announce completion.
3. Gather all rules whose consequent satisfies the current goal.
4. Consider each of the rules in turn:
   a. If all premises are satisfied (i.e., the value of each premise of the rule is defined in the database), then fire this rule to derive its conclusions. Do not consider any more rules for this goal. Its value is now given by the current rule's conclusion. If the goal presently being traced is a top-level goal, then

remove it from the stack, and return to step 2. If it is a subgoal, then remove this subgoal from the stack and return to the processing of the previous goal that was temporarily suspended.

b. If a value for a premise is found in the database, but the database value does not match the premise value, this rule fails to execute.

c. If any premise is not satisfied (that is, the premise value is not defined in the database), check for other rules whose consequent can derive a value for that premise. If such rules exist, then consider this premise value to be sub-goal, temporarily suspend the execution of the current rule, push the parameter onto the top of the stack, and go back to step 2 recursively.

d. If step 4c is unable to find any rules to derive the specified value for the current parameter, ask the user to enter its value and add it to the database; then go to step 4a and consider the next premise of the rule.

5. If all rules that can satisfy the current goal have been attempted and all have failed to derive a value, then this goal remains undetermined. Remove it from the stack and go back to step 2 [7].

In general the backward chaining algorithm looks first to the database for information, then to other rules that may be able to derive the information, and finally, as a last resort, asks for input from the user. Instead of initially observing facts about an object or situation and specifying all the facts up front, backward reasoning allows the system to ask the user for facts when they become important in deriving results. This approach often proves to be more effective since the user is not burdened with attempting to enter all information about the situation that might prove useful.

Backward reasoning uses the same rules in the same format as forward reasoning, but the rules are looked at backwards. The consequent of the rule is checked first to see if this rule is meant to solve the goal at hand. If it is an applicable rule, the premises in the antecedent are checked to see if they can be satisfied with available data. Since REFINE only provides built-in support for forward reasoning, a backward reasoning inference engine had to be built to support some of the EH capabilities. This inference engine is discussed in later chapters.

2.6 Data Dictionary

The data dictionary has become a very important tool for helping developers build applications from database definitions. Relational databases are currently the most popular method for modeling data and there

are many commercial database management systems (DBMS). One problem in developing applications from complex databases is that the developer has a very steep learning curve in understanding the database. A developer who has not been on the project during the data modeling phase must ask others who are more knowledgeable about the data to explain the data and relationships to him, or read through the development documents, which aren't always kept current. Therefore a large need exists for an interactive tool that can help the developer understand the database components and how they relate to each other. In a technical report about an intelligent Information Dictionary [13], a tool is described that implements a graphical user interface (GUI) using a hypertext type approach. The tool allows a user to mouse click on database table or relation between tables to see a description of the item. The user can also look at a list of the columns (attributes) of a table and view the description of a data column including the data type, unit of measurement, information source, constraints, and code meanings. By clicking on a relationship the user can view the *to* and *from* tables as well as the cardinality between them. As the user clicks on items of his choice, the tool harvests information from the database meta-data and the data dictionary tables and displays the retrieved information in a separate window.

The AFIT domain model has many conceptual similarities to a data model. Since data dictionaries are typically a standard part of database developments and are useful in helping to identify the data elements, it makes sense to provide similar capabilities to a knowledge-based system. The benefits of interactively harvesting information from an AFIT domain tree and its associated dictionary should mirror those benefits for database applications. Since the domain model stores more than just data information, such as states, events, and operations, it may be better to refer to the dictionary as an object dictionary, domain dictionary, or a knowledge dictionary. This paper refers to it as a data dictionary, since most readers are familiar with that term. An EH is itself an interactive tool that must elicit specification requirements from the application engineer and harvest knowledge existing in the domain model. During the specification process, the application engineer will certainly need to view domain elements in a context that can be understood in order to make decisions about the application under development. Having a data dictionary can provide descriptive information about classes, attributes, associations, operations, states, events, and transitions to assist in the interactive communication between the EH and user.

2.7 Eliciting Information Through a User Interface

One of the most difficult problems in software engineering is how to extract knowledge from users in a way that doesn't burden them with too many details, confuse them with a complicated set of instructions, or restrict them so much that they give up. Since the EH is a computer controlled process that needs extensive interaction with a human user, an approach to the human-computer interface needed to be studied. As with any interactive tool, an EH needs a well-designed user interface. Designing an effective interface is as much an art as a science and volumes have been published on the subject. The book *The Art of Human-Computer Interface Design* makes it clear how dissatisfied many computer users are with the existing interfaces [21]. Although it seems as though the jump from text-based to graphical interfaces was a big improvement, Theodor Holm Nelson, a software designer, writes: "Featuritis is a principal and well-known disease of software...You face a screen littered with cryptic junk...you try to understand what the icon means...The disease of featuritis is the unclarity and confusion that results from having too many separate, unrelated things to know and understand [21]." These opinions suggest that more pictures are not necessarily better than a few words if they are not designed intuitively.

The user interface will certainly be a very important part of the EH tool. At the very least it should guide the user through the process of building the specification in a way that doesn't cause the user to get lost or confused. It should allow enough latitude so the user can move around within the tool to view the progress, save an interim specification, or make corrections.

2.8 Other Relevant Research

There is current research in the area of specification refinement. The University of Hawaii is currently working with an Army interactive combat simulation system called ModSAF (Modular Semi-Automated Forces [10]. Their work focuses in refining high-level specifications into more detailed specifications which can be formed into executable simulation scripts. Many of their refinement tasks are analogous to EH tasks. Their refinement process takes a user's high-level exercise training specification, a domain model of movement operators, and a database with terrain descriptions as input and produces a detailed exercise specification meeting the requirements as output. The domain model is basically a set of rule operators that add to the refined simulation script when all preconditions are met.

Their approach is a plan-based AI technique being applied to a very restricted domain. Even with a very restricted domain, the potential for combinatorial explosion of possible rule searches exists. They therefore used a hierarchical planning scheme that maintains rules at different levels of abstraction and operates on those subsets of rules in a controlled order.

An approach like this may be useful for guiding an application engineer through the specification process with the EH. Essentially, the EH is a tool to help the application engineer refine the knowledge stored in a domain into a more descriptive set of specifications. Theoretically, it should be possible to create an automated process to transform the domain description into a low-level software specification with minimal human effort by dividing rules into related sets and breaking down the process into smaller sub-processes that look at a certain part of the domain model. It should be reasonable to start with more high-level specifications, such as identifying operations, classes, and associations required, and work down to the very detailed specifications like data type definitions, constraints on attributes, and multiplicities of associations.

During the specification refinement process, it is often necessary or preferable to change or transform the structure of the object model into a different form that better fits the specification. Several data model transformations are explained in a paper by Blaha and Premerlani [12]. Some of the transformations explained include adding or removing constructs, restricting multiplicity, partitioning or merging constructs, composing associations, moving attributes among generalizations, and various other inheritance manipulations. When such a transformation takes place, the new form should be equivalent to the old form and no information should be lost. An EH user may not necessarily understand the need or reason for these transformations, so heuristics can be applied to find the needed transformations and recommend them to the user.

3 Requirements

The purpose of this chapter is to define the requirements of an EH (Elicitor-Harvester) process that would fit into the framework of the object oriented knowledge base system being studied at AFIT. First, Section 3.1 discusses the thought processes that went into defining the requirements. Next, the inputs and outputs of the EH are described in the Section 3.2. The largest part of this chapter, Sections 3.3 and 3.4, focuses on the requirements of the functional capabilities, which include the actions allowed during the specification phase; the restricted or prohibited actions; and the clean-up process, which purges the unnecessary parts remaining in the specification. Section 3.5 discusses the interaction requirements between the user and the EH along with various possible approaches to a user interface. Following the *User Interface* Section is a section about the benefits and uses of AI (Artificial Intelligence) techniques in the EH process. Additional domain and specification AST (Abstract Syntax Tree) components required by the features of the EH are described. Finally, the sample domain models used for explanation, implementation, and testing are discussed.

3.1 Philosophy Behind the Elicitor-Harvester Requirements

The concept of an Elicitor-Harvester is still fairly new, so the requirements and expectations of such a process have not yet been well defined. As KBSE matures and commercial tools become available, the capabilities of an EH will certainly evolve so it smoothly integrates into the overall framework of the knowledge based tool set. The requirements described in this chapter are somewhat limited in scope to support *this* thesis effort. There is certainly room for further improvements and refinements that would improve performance or ease the use of an EH tool. Some suggested improvements are discussed in the Recommendation section of Chapter 6.

Despite many philosophical discussions, it is difficult to agree upon what should or should not be allowed during the specification process. It is generally agreed that the EH process should not allow changes to the domain description, otherwise any application engineer could override the decisions of the domain expert, thus affecting future specifications built from that domain. On the other hand, it may be too restrictive to prohibit the application engineer from creating new objects not included in the domain, but required for a specific application. It also may be too restrictive to prohibit the modification of domain components into a form more suitable for the specified application. It is certainly not reasonable to expect a domain expert to have

enough foresight to anticipate every possible event or data element that may be needed for all future applications.

One way to help protect the domain and allow flexibility for the application engineer is to create a distinct specification AST that is separate from the domain AST for each application developed. The application engineer would then have the freedom to modify existing objects and operations and create new ones where necessary as long as those changes do not violate constraints defined in the domain.

Previous EH thesis studies at AFIT, Cochran [6], Wright [3], and especially Karagias [2], have attempted to define specific changes and transformations that should and should not be allowed during the specification phase. While this is certainly a desirable long range goal, the KBSE field is still too immature to nail down all the specific requirements of an EH tool. Since the specification phase is in the middle of the KBSE process, it is affected by the completeness of the domain description and it has the ability to cause problems downstream in the design and implementation phases. In relation to the domain AST, the EH should allow the application engineer to add or modify domain components if the knowledge represented in the domain model is not adequate to support the specification requirements. It should also be possible to clean up the specification AST by deleting unnecessary attributes, constants, data types, etc. at the end of the specification phase.

With regard to the design and implementation phases, changes made to the model during specification could affect interface issues between resulting applications. For example, say an application engineer selects an object class named *person* from the domain model and uses it as is for application 1 (A1). A second application engineer creates application A2, which also uses the *person* class, except the engineer removes the *address* attribute and changes the data type for *age* to *Months* instead of *Years*. Later, it is decided that A1 and A2 should interface with each other; but because the two data structures storing the *person* records are different, the *person* records could not be easily merged. Even though both applications started with the same structure for *person*, some translation functions would now be needed between the two applications.

It is probably impossible to prevent all scenarios of this type, and there is obviously a trade-off between specialization and standardization. The final decision between the two will depend on the management philosophy of the organization using the KBSE system. The multitude of such issues begs for an approach to specification that allows as much latitude as possible without violating deliberate domain constraints. Therefore,

an EH tool should provide great flexibility for the application engineer and leave the standardization issues to configuration management.

A production EH tool should provide a knowledge base administrator the capability to define restrictions to be placed on certain specification functions. Similar to how a database administrator can set access privileges for database users, the knowledge base administrator could set the preferences of an EH to allow or prohibit certain specification abilities. With these thoughts in mind, the following sections define the requirements and scope of this version of an Elicitor-Harvester.

3.2 Input and Output Requirements

A good way to begin describing a process is to define the inputs and the outputs. The following paragraphs first look at the inputs, which consist of the domain AST, rules, the data dictionary; and inputs from the application engineers, and finish up with the outputs, which include the specification AST, user messages, data dictionary updates, and history database. Figure 8 shows the operating environment of the EH.

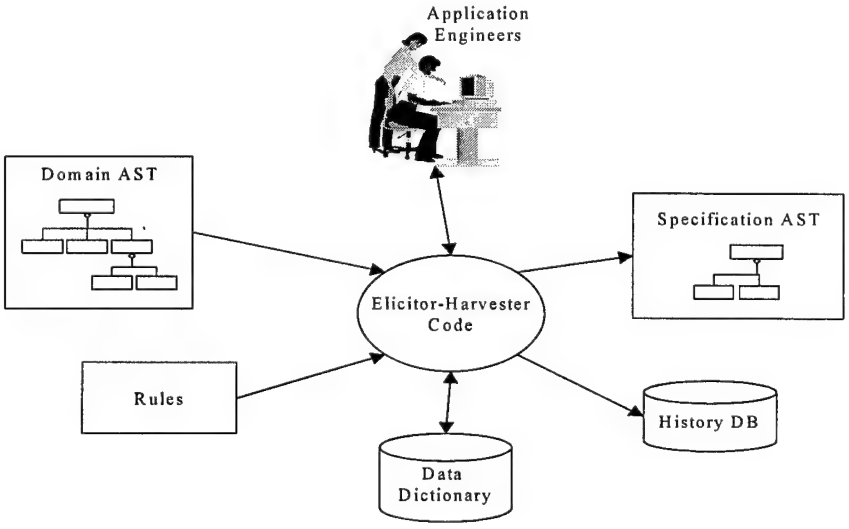

Figure 8 Elicitor-Harvester Environment

3.2.1 Elicitor-Harvester Inputs

The EH receives inputs from the domain AST, the users or application engineers, rules, and the data dictionary. These inputs are explained in the following paragraphs.

3.2.1.1 Domain AST Input

The main input to the EH is a domain AST, which stores knowledge about a particular domain. The representation used by the domain AST is generic enough to handle most types of domains. Although the sample domains available at AFIT are not complete domain descriptions, it is assumed that the domain being operated on by the EH is a reasonably complete model that has been thoroughly defined by a Domain Engineer with the aid of a Domain Expert.

3.2.1.2 Application Engineer Input

The application engineer, who will be called the *user*, is the person who operates the EH for the purpose of creating a specification for a new application. The EH communicates with the user through a user interface, which is discussed more thoroughly later in this chapter. The EH queries the user for information about the new specification and the user provides that information by keying in a response.

3.2.1.3 Rules

The rule base supplies the EH with derived values based on the objects passed to the rules and the data available in the domain and specification ASTs. Input from rules is usually indirect. A rule, if executed, generally updates some object or a field in a data structure, which is called a *fact base*. The EH can access those updated items and act on the new information accordingly.

3.2.1.4 Inputs from the Data Dictionary

The data dictionary stores aliases for names of components declared in the domain. Since a user may not know the names of the objects in the domain, the responses keyed in may not match the object names. If the user enters a term not recognized as a domain identifier, the EH queries the data dictionary for synonymous terms that may match the user's input. Since domain names are often abbreviated, the EH uses a set of rules or heuristic functions to generate reasonable abbreviations, which often help match user input to domain names. During the domain-engineering phase, domain items (data elements) should have their descriptions and synonyms stored in the data dictionary section of the domain.

3.2.2 Elicitor-Harvester Outputs

The outputs of the EH include a specification AST for a particular application, a history database, and new entries to the data dictionary. Its outputs must also include informative messages and data requests to the application engineer via the user interface.

3.2.2.1 Specification AST

The main output of the EH is the specification AST. If the specification is defined properly and thoroughly, it should allow for an efficient automated transformation to the design phase. The specification tree should contain all associations, classes, attributes, attribute restrictions, operations, pre and post conditions, events, states, transitions, data types, and constants. Of course, not all details are specified at this stage. The specification identifies *what* needs to be done, but not necessarily *how* to do it. Particular data structures, file formats, algorithms to optimize performance, and temporary variables are a few examples of software that do not get defined until the design phase.

There are two approaches to creating the specification AST. One way is to build it from the ground up by adding components as they are identified as necessary to the specification. The second way is to make a copy of the entire domain AST, make required changes, and then prune the parts not necessary for the specification. A disadvantage of the first approach is that it requires the maintenance of two ASTs during the EH process – the harvesting of the domain tree, and the building of the specification tree. The advantage is that the clean-up process is simpler because only a subset of the domain AST is added to the specification AST. The advantage of the second approach is that it only requires one AST (the copy of the domain AST), which is augmented with new components then pruned of the unnecessary items. Although the pruning (during clean up) is more extensive, the advantage of working with a single AST that contains everything probably outweighs the disadvantage of extra pruning. Here is the subtle kicker: as the application engineer adds to and modifies components in the domain model, the specification AST begins to contain items that should be checked and harvested for further refinements during the specification phase. By using two ASTs, the EH must perform searches on both the domain AST *and* the specification AST while harvesting for knowledge. This (first) approach can become an implementation nightmare. For this reason, the second approach was chosen. Objects in the AST need an extra flag or two to indicate when they have been selected as a required part of the new specification. These flags are also used during clean up to help identify which parts to prune from the tree.

3.2.2.2 History Database

During EH processing, many decisions are made and many actions performed. There may be times when the user wants to change or delete one or more of those actions, similar to how an "undo" function works. There needs to be a way to log the activities performed during the EH process so that each time a specification item is selected, added, modified, or deleted, the description of that action is stored. The *history database* fulfills these requirements. It is a repository of EH activities stored in such a way that a sequence of actions can be viewed for editing. The long-term aspects of such a feature would allow a user to go back and undo specific actions by selecting from a list of prior actions. An entire specification could also be replayed, which would allow the application engineer to fine-tune the specification. Imagine, for example, that an application engineer has completed the specification process and finds out only a few minor changes are needed. The application engineer could access the history database, modify a couple of historical actions, and rerun the specification process from the history database without having to go through the entire EH process again.

For this research, the history database has been limited to simply storing in chronological order the actions that caused a change to the specification AST and providing a mechanism to view it. Other functionality should be studied and is discussed in the recommendations section of Chapter 6

3.2.2.3 Output to the Data Dictionary

When the user keys in identifying names of domain tree objects desired for the specification under construction, there are cases when no matching synonym is found in the data dictionary. It is desirable for the EH to have the ability to add new synonyms to the data dictionary for future use. The EH would appear to "learn" new synonyms for the domain objects. This learning functionality is outside the scope of this research and is discussed in the Recommendations section of Chapter 6.

3.3 Functional Requirements

Now that the input and output requirements have been defined, the following sections discuss the functions allowed and prohibited during the transformation from domain AST to specification AST.

3.3.1 Operational Capabilities

This section outlines the actions an EH should be able to perform. In describing capabilities of the EH a method similar to one describing DBMS (Database Management System) functions will be used. The actions

allowed on database tables are Create, Retrieve, Update, and Delete (CRUD). These terms, which have been borrowed and changed to Create, Select, Modify, and Delete, describe the actions allowed on AST components and are defined below. As mentioned before, a knowledge base administrator should be able to set the restrictions for many of the functions depending on the management philosophy of the using organization. Therefore, an additional term called "Restrictable" is also defined. A generic term is needed to identify a part of the AST that could refer to a class, attribute, state, constant, data type, or any other part of the domain. Since the word "components" can lead to ambiguity, the word "item" is used instead.

- **Create** – Ability to create a new item that needs to be added to the specification.
- **Select** (Retrieve) – Ability to use the item as defined in the domain model. All items are assumed to have the ability to be selected unless otherwise stated.
- **Modify** (Update) – Ability to change the item by applying some transform to the item such as merging or generalizing as described by Blaha [12]. Modify also includes adding, deleting, or changing anything in the subtree of the item. For instance, if the data type of an attribute within a class is allowed to be changed (not restricted), then that is considered an allowable modification to the class.
- **Delete** – Ability to remove an item from the specification tree during specification or clean-up functions.
- **CAUTION!** If not performed properly many *Delete* functions can cause problems because the item being deleted may be referred to in other parts of the domain. Therefore, these functions should be used at the end of the specification phase during the clean-up process to ensure necessary items are not deleted in the specification phase.
- **Restrictable** – This means an item can be restricted from create, modify, or delete actions by the knowledge base administrator. Management may decide certain functions should be restricted to maintain consistency between different applications developed from the same domain.

3.3.1.1 Primitive Classes

- **Create** – New classes can be created; however, this may be a restricted function if the domain is considered complete. In some cases, a new class can be created as a sub-class in an inheritance tree. This approach is a good way to add attributes or items to a class, which may be the preferred alternative to changing an existing class.

- **Modify** – Classes may be modified by changing items in the class subtree. These changes can include adding, modifying, or deleting the class name, type declarations, constants, attribute names or types, predicates (invariants), operations, states, transitions, or events. Modification of some or all items may be restricted.

- **Delete** – Classes may be deleted from the specification AST if they are not used in the specification or if the deletion results from merging two classes in a transformation. This function may be restricted.

3.3.1.2 Class Attributes

- **Create** – New attributes can be added to a class. This function may be restricted if the chosen alternative for extending classes is to inherit new sub-classes.

- **Modify** – The name or type of a class attribute can be modified. Modify functions can be restricted.

- **Delete** – Attributes can be deleted if not used in the specification.

3.3.1.3 Class Operations

- **Create** – New operations on a class can be created.

- **Modify** – An operation can be modified by changing the name; adding, changing, or deleting parameters or predicates (pre or post-conditions); and/or changing the sub-operations. These functions can be restricted.

- **Delete** – Class operations can be deleted if not used in specification.

3.3.1.4 States

- **Create** – New states can be created since new attributes are allowed to be created. The state of an object depends on its attribute values. This function can be restricted.

- **Modify** – States can be modified by changing the name, or by adding or changing the predicates (state invariants). This function can be restricted.

- **Delete** – States can be deleted if the attributes identified in the invariants have been removed from the specification.

3.3.1.5 Events

Since events are defined in classes, the mapping between sending and receiving events must be specified at a higher level. These event mappings will need to be defined at the domain level after all class events have been specified.

- **Create** – New events generated by an object or events to which the object or responds can be created. If the object must respond to the new event, states and transitions associated with the new event must also be created or modified.

- **Modify** – The name, parameters (arguments), and predicates (parameter constraints) may be changed. These functions can be restricted.

- **Delete** – Events can be deleted from the specification tree if not applicable, but great care needs to be taken to assure associated states and transitions are properly handled.

3.3.1.6 Transitions

- **Create** – New transitions will need to be created if new states or events have been added.

- **Modify** – Transitions can be modified by changing the predicates (guard conditions) or operations (actions). This function can be restricted.

- **Delete** – A transition may be deleted if any of the following do not appear in the specification: *caused-by-event*, from-state, to-state, attributes identified in the predicates. It may be wise to produce an error message if items are missing from the transition sub-tree.

3.3.1.7 Parameters

- **Create** – New parameters can be created when added to the parameter list of an operation or event.

- **Modify** – The name, type, and output flag can be changed. Modify functions can be restricted.

- **Delete** – Parameters can be removed from the parameter list of operations or events.

3.3.1.8 Predicates

- **Create** – New predicates can be created when adding constraints to an event, guard conditions to a transition, invariants to a class, or pre and post-conditions in an operation. Restrictions may be placed on predicate creation and will depend on the restrictions to the items of which the predicate is a part.

New predicates should be checked for correctness to be sure that the variables in the predicate correspond to existing domain items.

- **Modify** – Since many predicates define constraints on the domain model, the only changes allowed to a predicate should be ones that further restrict the constraint for a particular application. It may be very difficult to ascertain whether a change to a predicate constitutes further restriction or not. Because predicates are quite complex and flexible, the AI abilities would have to be very sophisticated to decide if a change resulted in further restriction. This degree of sophistication is beyond the scope of this research, therefore the designer of an EH may choose to prohibit modification of constraint predicates. In some cases, constraints can be implemented by restricting the range of a data type, which may be preferable over a predicate. Changing pre-conditions and post-conditions in operations is allowed in order to allow refinement of functionality. Modified predicates should also be checked for correctness to be sure the variables in the predicate correspond to existing domain items.

- **Delete** – A predicate can be removed during clean up if none of the items named in the predicate remain in the specification. Predicates defining domain constraints should not be deleted if applicable to the specification.

3.3.1.9 Data Types

Data types can be defined at the global level (*dom-global-type* attribute of *GOMT-DomainTheory*) or locally within the class (*dom-private-types* attribute of the *GOMT-Class*). An application engineer should have the ability to move data type definitions between the local and global levels if desired. If more than one class declares the same data type, it would make sense to move the type declaration up to the global level. On the other hand, a global data type should be moved down to the class level if it is only used in one class.

- **Create** – New data types can be created. It may be useful to create derived or sub-types to help define constraints. These definitions can sometimes replace predicates such as class invariants.

- **Modify** – Data types can be modified by changing name, the enumerated list of type values, the predicate (usually the data range), and type multiplicity. Data types are often named but not defined in the domain model, and therefore should be defined more clearly in the specification if possible. These functions can be restricted for certain data types, which the domain engineer wants to keep as is. See Section 3.3.2 for more details.

- **Delete** – Data types can be deleted during clean up if not needed in the specification.

3.3.1.10 Constants

Constants, like data types, can be defined at the global or class level; the same issues mentioned under data types also apply to constants. It is possible that two or more constants could have the same name and may or may not have the same meaning. The EH will depend on the application engineer to decide whether to move constants between global and class levels.

- **Create** – Constants can be created if the application engineer finds it useful.
- **Modify** – Constants can be modified by changing the name, type, or value. Since constants can be declared in the domain model with just a name, the type and value can and should be added during the specification phase if possible. These functions can be restricted for certain constants the domain engineer wants to keep as is. See the Section 3.3.2 for more details.
- **Delete** – Unneeded constants can be deleted from the specification during clean up.

3.3.1.11 Inheritance

Inheritance can be used as a method for extending class definitions by adding a new sub-class instead of modifying the existing class. This alternative may be preferred in cases where items need to be added to a domain class, but management wants to maintain consistency of the domain class definitions.

- **Create** – New sub-classes that inherit all items from the super-class can be created, and can have more items added to them in order to meet the specification requirements.
- **Modify** – Many transforms can be applied to inheritance structures [12]. An abstract super-class may have several sub-classes representing choices for the application engineer. Once the choices are selected, the abstract class may no longer be necessary and can be deleted from the tree leaving just the chosen sub-class in the specification. Class attributes or operations can also be moved between the super-class and sub-classes, but this function can be restricted.
- **Delete** – An inheritance class can be deleted if not needed in the specification.

3.3.1.12 Associations

- **Create** – Associations between classes can be added to allow description of relationships that were not thought of or included in the domain model. New associations must have the multiplicities defined, as well as attributes if necessary.

- **Modify** – Attributes of the association can be added, modified or deleted. Multiplicity of existing associations can be tightened, but not made less stringent. See Section 3.3.2.

- **Delete** – An association need not exist between two classes in the specification if not required in the application, so it can be deleted.

3.3.1.13 Aggregate Classes

Aggregates are classes that contain other classes. Aggregation is a special type of association that models the *has-a* relationship; i.e., the class has a component. The relationship is modeled in the AST with the has-components and has-aggregate-associations attributes. Aggregates may also include regular association definitions between classes in the domain. These classes are sometimes called system classes.

- **Create** – New aggregates can be created. The new aggregate can be the result of assembling several parts of the domain, or creating a sub-class of an aggregate class, which inherits the components of the parent class, plus adds more items to further specialize the aggregate.

- **Modify** – Aggregates can be modified, since the aggregate is made up of many other items that can be modified. Classes, associations, and operations can be added to or removed from the aggregate as long as no prohibited or restricted functions are performed.

- **Delete** – Aggregates can be deleted from the specification if not needed.

3.3.1.14 Aggregate Operations

Whereas primitive class operations have limited scope to the class in which they are defined, aggregate operations can perform actions on multiple classes. The EH tries to identify aggregate operations early in the specification process in order to infer the classes and associations needed for the specification.

- **Create** – New aggregate operations can be created.

- **Modify** – Aggregate operations can be modified by changing the name; adding, changing, or deleting parameters or predicates (pre or post-conditions); and/or changing the sub-operations. These functions can be restricted.

- **Delete** – Aggregate operations can be deleted if not used in specification.

3.3.2 Prohibited and Restricted Actions

With the EH philosophy in mind, the actions restricted or prohibited for this EH are outlined.

- Predicates in classes often describe invariants intended to restrict attribute value ranges. When these constraints are restricted, they should not be allowed to be relaxed during the specification phase.

- Multiplicity of associations defined in the domain cannot be expanded. E.g. a one-to-one relation cannot be expanded to a one-to-many relation. An association can be constrained further. E.g. a zero-or-one-to-many can be re-defined in the specification to be one-to-three.

- Constants with defined values generally should not be changed and can be restricted. However, if a constant with the same name but different values or data types appears in more than one class, the user will be allowed to move the constant to the global level and choose between the two values.

- Constants, data types and other items are sometimes carefully defined by the Domain Engineer with the intent that the item should not be changed during future phases. Such cases can be identified in the domain model to prohibit the EH from changing those items. See Section 3.7 for recommended AST changes to support this requirement.

3.4 Clean-up Process

After the application engineer finishes with the specification, many parts of the AST that are unnecessary for the application being specified still remain. These extra parts are removed from the specification AST during the clean-up process. Depending on the kind of item being removed, the EH checks the parts of the domain that may use the item to make sure a required item is not deleted. During the clean-up process, the user is asked to verify the proper declaration level for data types and constants (global vs. local). If inheritance structures exist, decisions about abstract vs. concrete classes and placement of their attributes are made. After events have been cleaned up and only required events remain, the mappings between the *from* events and the *to* events are made.

3.5 User Interface

Ideally, the user interface should help the user be as efficient as possible. Efficiency means that the user gets a lot of work done with a small number of key strokes and mouse clicks. The interface should guide the user through the specification process in such a way that the user is not confused and the EH gets the information needed. The most common and accepted way of attaining efficiency in modern applications is with a Graphical User Interface (GUI), with which a user can make selections from menus and lists by clicking the mouse cursor on the selected items. Of course, many GUI development tools exist today and the Intervista GUI

software is provided with the REFINE package, but developing GUI interfaces can be very time consuming. Since the main thrust of this research of building a smarter EH methodology does not require a GUI, and after considering the time needed for the various options, a text-based interface was chosen. The basic requirements of a text-based interface are the ability to accept user input from the keyboard and print important information to the screen.

3.6 Artificial Intelligence Techniques Employed

AI techniques are utilized in the EH to help remove the burden of the many of the specification details from the application engineer. AI is mainly manifested in rule-based reasoning and heuristic functions. Rule bases have been created as a way to apply actions to the specification tree. Because there are so many different types of objects in the AST, trying to uniquely process each type of object would cause the source code to get very messy. Rule bases help simplify the code and keep it more structured. Heuristics are "rules of thumb" that are applied to infer the best action from the given circumstances. Heuristics have been implemented in the rule base and in algorithmic functions. The representation of the knowledge base as an object oriented AST and the search methods used over the tree also fit into the category of AI. These techniques are used in the EH to match terms in the data dictionary, select objects from the domain tree for use in the specification tree, make recommendations to the user, validate modifications and check restrictions, and help perform the clean-up functions.

3.7 Modifications to ASTs

Additions to the domain and specification AST were determined to be helpful to the EH process.

- Section 3.3.2 describes how some items of the domain need to be marked as restricted from change. To support these cases an extra attribute is needed in the *GOMT-Object* class to indicate whether this particular item is allowed to be changed or not. The Domain Engineer can set this attribute to prohibit the EH from making changes to an item.

- The data dictionary components are needed in the domain to provide the ability to store the string name of the domain element, multiple synonyms, a description of the domain element, and a pointer to the domain object to which the element refers.

- As mentioned in the last paragraph of Section 3.2.2.1, an additional boolean flag is needed for all *Obj-Objects* to indicate whether this item has been selected for use in the specification.

- A *Library-Ops* class is needed in the domain AST to accommodate pure operations (general operations that don't belong to any particular class in the domain, such as math functions like *SquareRoot*(x)).

3.8 Sample Domains

In order to test the EH functions, an actual sample domain and specification should be used. It is preferable to use as close to a real world example as possible. During previous research at AFIT, domains modeled in object-oriented architecture have tended to fall into different categories. Domains that model something to be built are aggregation oriented, such as the CRUISE MISSILE domain, which consists of the various parts of a cruise missile. Some domains are biased toward states and events, such as the TRAFFIC LIGHT INTERSECTION model. Some models consist of many primitive classes related through associations with domain level operations acting on multiple classes, such as the SCHOOL and TRAIN domains. Since it was difficult to come up with a single domain that could exercise all parts of the EH, it was necessary to use two domains as examples. The AFIT KBSE group had several incomplete domains already defined. Since creating a new domain can be very time consuming, the pre-existing AFIT domains were used. The incompleteness of the domains actually helped this research because it was easy to come up with examples for most types of additions and modifications.

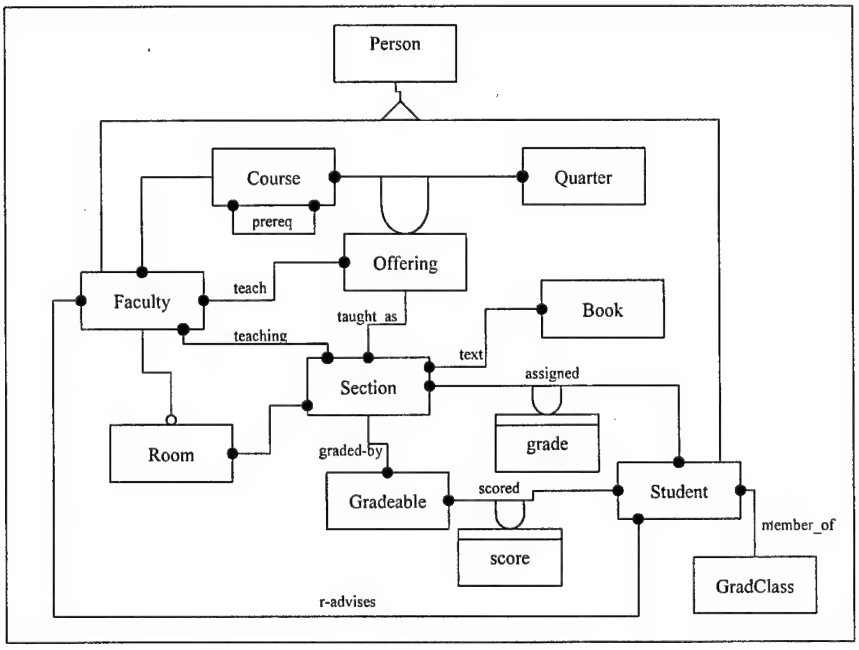

Figure 9 Classes and Associations for the School Domain

The SCHOOL domain, shown in Figure 9, was chosen as one example domain since it is fairly extensive and there existed a well-defined specification for a scheduling application. The SCHOOL domain was used to test the actions performed on primitive classes, and their attributes, constraints, constants, data types, and operations. The SCHOOL domain helped exercise the inheritance functionality since *Faculty* and *Student* classes are subtypes of the *Person* class. It was also used for associations and pure operations. Since there is no *has-parts* aggregation in the SCHOOL domain, the CRUISE MISSILE domain, shown in Figure 10, was used when aggregate manipulation functions needed to be tested. For more detailed information about the sample domains, refer to the Z-schemas in Appendix B.

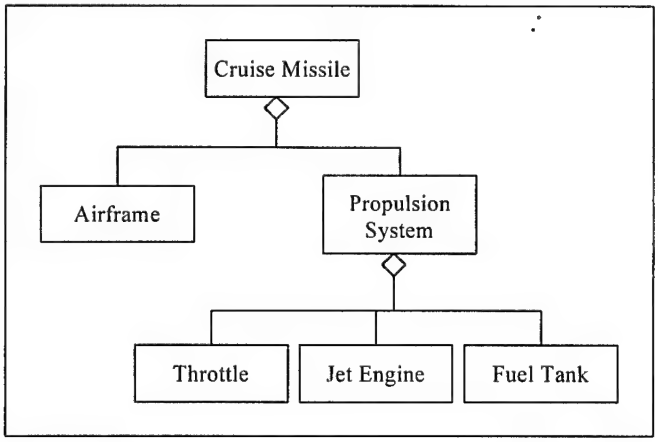

Figure 10 Class Hierarchy for the Cruise Missile Domain

3.9 Requirements Summary

This chapter described several components required for an intelligent, user-friendly Elicitor-Harvester tool. The domain AST, data dictionary, rule bases, inference engine, fact bases, and history database all need to be orchestrated within the user interface to provide the user with a clear and intuitive way to build specifications. The input and output requirements were defined along with the actions allowed on the various types of domain items. Chapter 4 discusses the approach used to integrate the components of the EH into a useful, interactive tool. The design details used in building the EH tool are described along with the philosophy behind them. Chapter 5 discusses the functionality actually implemented, some of the difficulties encountered during development, and an evaluation of the implemented tool.

4 Design

Since the area of eliciting and harvesting of a domain model is still in the early stages of research, the development of an EH (Elicitor-Harvester) tool could not follow the traditional waterfall method of the sequential requirements, design, implementation, and testing stages. The requirements in Chapter 3 were kept fairly general because it was not known if this research effort could meet those requirements. Approaches to user-computer interaction were very vague, and attempts to lay out a design usually brought out more questions than answers. Given this scenario, it would have been imprudent to try to completely design an automated EH process without knowing whether many of the details would work. As a better alternative to a waterfall approach, it was decided that iterative prototype development would be more effective.

The main philosophical point kept in mind during each design decision was always to try to use the knowledge built into the domain or try to infer information from the domain before asking the user for input. The EH should take as much burden off the user as possible by limiting the number of choices when an input is requested. This chapter discusses the approach used to design the EH.

4.1 Data Dictionary Design

An application engineer may not always know the names of all domain objects, so when asked to enter the name of a domain item, the user may type in a different name. Further complicating the problem are the various methods of naming identifiers. Names are often abbreviated (*prod_ID* instead of *product_identifier*) and multiple word identifiers may have no space between them or may have underscores, hyphens, or dots (for example, prod-id, prod.ID, ProductID). Realizing that identifier ambiguity could pose a problem during user interaction with the EH, the use of a data dictionary was considered. This section discusses the design of the data dictionary structure, the associated rules, functions, and fact base, and its uses.

4.1.1 Data Dictionary Structure

The AFIT KBSE system had no data dictionary capabilities except an unused *description* attribute that could be used to store a text description for each object. Therefore, to aid the application engineer during the specification process, a simple data dictionary structure, as shown in Figure 11, was built to store data elements corresponding to domain items. As part of the feasibility evaluation, functions were written to populate the data dictionary with the data element name, synonyms of the name, an optional description, and a pointer to the

actual domain object described. Ideally, a data dictionary would be populated during the domain-engineering phase. The system should be able to automatically create a data element for each domain item, but the domain engineer should supply descriptions and synonymous terms for each item. Later, during EH processing, the data dictionary can be searched to help match names entered by the application engineer to domain objects.

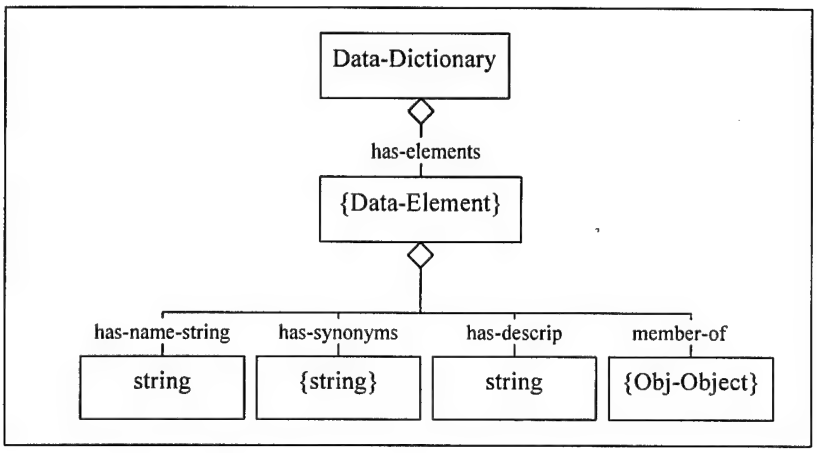

Figure 11 The structure of the Data Dictionary class

4.1.2 Handling User Inputs

To help match names to actual domain objects, a small set of rules was written to act on a fact base called *Aword* as shown in Figure 12. When the EH needs to match a word from the user's input to a domain object, the set of match rules are used to find objects with similar names. When the user inputs text, the EH

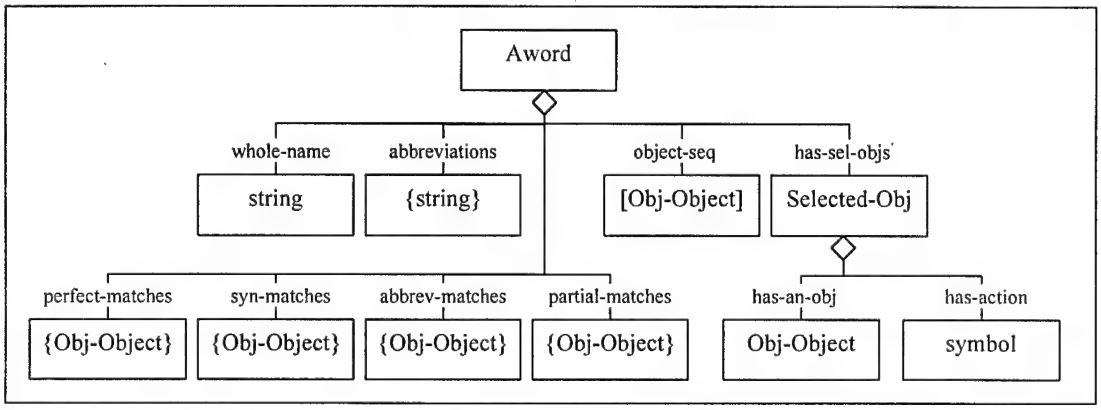

Figure 12 The Aword structure. The fact base used when matching input names to domain objects

must parse the text string and store it in the *Aword* structure for further processing. The string of text, which can be a single word or multiple words, is read into a string and passed to the *String-to-Seq* function. The *String-to-Seq* function removes commas from the string, separates the words into a sequence of strings, then

stores them in a sequence of *Aword* objects called *Phrase*. Some text processing functions, such as *String-to-Seq*, employ calls to Common Lisp functions since REFINE is not very rich in string and character manipulation capabilities. The REFINE system is generally case insensitive, but when trying to match strings, the case becomes important.

Once the user's input has been parsed into the sequence of *Aword* structures, the EH calls a *preorder-transform* function to traverse the data dictionary while applying rules to the data elements. The function essentially searches the data dictionary for domain object names that match the input word in various ways. There are four basic ways the input can match a data element. Ordered from most desired to least desired, these possible matches are: a perfect match, a synonym match, an abbreviation match, or a partial match and their descriptions are given here:

1. Perfect Match – The input string exactly matches an element name in the has-name-string attribute of the data dictionary.

2. Synonym Match – The input string exactly matches a name from a has-synonyms attribute of the data dictionary.

3. Abbreviation Match – An abbreviation created from the user input matches the name or synonym of a data dictionary element. The abbreviations are created from the user input by using the first three, four, five, or six letters of the word (if the word is at least two letters larger than the abbreviation); and an abbreviation is created by removing all vowels except for a leading vowel. All abbreviations must be at least two characters long.

4. Partial Match – The user input or abbreviation matches a sub-string of a name or synonym in the data dictionary. The Lisp function called Search is used to make these comparisons, and each time a sub-string is matched, a counter is incremented, which can be used to measure how good a match it is.

When a matching data dictionary element is found, the domain object that is pointed to by the *member-of* attribute of the data element is added to the set of matching objects in the *Aword* structure. Each object in the matched sets in Aword are candidate objects for the user with the perfect matches being the most likely candidates and the partial matches being the least likely.

4.1.3 Using the Data Dictionary

Often during processing the EH will request information from the user such as the name of a domain item needed for the specification or a predicate for a new operation. The EH needs to ensure that the objects it

selects are indeed the same objects the user wants for the specification. The function *Match-Word* is called which creates the *Aword* structure and starts the rule-processing engine. The rules add objects to the *Aword* structure that match the user input in the four ways mentioned earlier. The set of matched objects is passed to the *Sequence-Objects* function where they are sequenced by object type in preparation for a pretty print to the screen. The list of objects is passed to the *Print-Obj-List* function, which prints the list of candidate object matches to the screen as shown in Figure 13, and asks the user to choose the proper object.

```
Enter one of your desired OUTPUTS or return key to return to Main Menu
fuel_level

(0) None of these
(1) CLASS: FuelTank
      HAS-ATTRIBUTES: tank_sim_time;  input_flow_rate;  output_flow_rate;
        fuel_level;  capacity;  tank_weight;  fuel_density;
        fuel_tank_weight;
(2) CONNECTION: JetPropulsionSys.fueltank : FuelTank
(3) ATTRIBUTE: FuelTank.fuel_level : Real
(4) ATTRIBUTE: Throttle.actual_flow_rate : Real
(5) ATTRIBUTE: Throttle.maximum_flow_rate : Real
(6) ATTRIBUTE: JetEngine.current_fuel_flow_rate : Real
(7) ATTRIBUTE: JetEngine.maximum_fuel_flow_rate : Real
(8) ATTRIBUTE: FuelTank.fuel_tank_weight : Real
(9) ATTRIBUTE: FuelTank.fuel_density : Real
(10) ATTRIBUTE: FuelTank.output_flow_rate : Real
(11) ATTRIBUTE: FuelTank.input_flow_rate : Real
(12) ATTRIBUTE: JetPropulsionSys.prop_fuel : Real
(13) OPERATION: DetermineFuelWeight
      HAS-PARAMETERS: fuel_weight;
(14) OPERATION: CalculateNetFlow
      HAS-PARAMETERS: net_flow_rate;
(15) OPERATION: InitFuelTank
      HAS-PARAMETERS:
(16) OPERATION: LoadFuel
      HAS-PARAMETERS: fuel_load;
(17) INPUT PARAMETER: flow_rate  IS: Real
(18) INPUT PARAMETER: fuel_weight  IS: Real
(19) INPUT PARAMETER: fuel_load  IS: Real
(20) OUTPUT PARAMETER: fuel_tank_weight  IS: Real
(21) OUTPUT PARAMETER: fuel_weight  IS: Real
(22) OUTPUT PARAMETER: net_flow_rate  IS: Real
(23) OUTPUT PARAMETER: overflow_event_time  IS: SIMTIME

Enter the number of an object you will want to to use for 'fuel_level'=>
```

Figure 13 List of domain objects matching the user input for *fuel_level.*

The user enters the item number desired and the number is returned to the calling function where the chosen object can be used as needed. Figure 13 shows many objects in the CRUISE MISSILE domain found to be possible matches for *fuel_level,* even though a perfect match was found in choice (3). The other objects were partial matches that had a sequence of at least two characters in common with *fuel_level.* These results demonstrate the usefulness and feasibility of a data dictionary. However, on large domains, this method of displaying all matches causes the list to be too long, which detracts from the ease of use of the interface. This

version of EH does not use the full power of the matching schemes, but simply lumps all four matching sets together when executing. A fuller version of the EH would check the matched sets in sequence in an effort to decrease the number of choices for the user.

4.2 User Interface Design

The interface functions as an integral part of many sub-functions of the EH, and as such cannot be described as a component or separate module of the EH tool. Since the interface is basically the user's window into the EH, the screens controlled by the interface are used to guide the reader through the functionality of the EH. Describing what EH does with the information keyed in and how it derives the information displayed on screen should aid the reader in understanding the EH process.

The user-computer interaction basically consists of the EH printing inquiries to the screen prompting the user to key in required information. The main purpose of the user interface is to methodically pull information from the user in a way that elicits input from the user only when the knowledge cannot be harvested from the domain model. The process should flow in a reasonable way such that the user understands what needs to be entered.

Several standard functions were built to give the screen displays a common look. Often the user directs the control flow of the process by choosing one option from a menu or list of actions printed to the screen, as shown in Figure 14. Other times the user is asked to pick from a list of objects to be acted upon, as shown in Figure 13. Two functions, *Print-String-List* and *Print-Obj-List*, take a sequence of strings or objects respectively and an informational message string as arguments. The functions print out a numbered list of choices, print out the message string, which usually instructs the user what to do, and return the number selected by the user. Choice (0) is consistently printed as "None of these" in all lists printed to the screen. By choosing (0), the user can usually back out of this screen gracefully if it's not where he wants to be or none of the other choices are satisfactory. *Print-Obj-List* calls the *Obj-Description* function, which formats a short description of each object in the list. Figure 13 shows the format that *Obj-Description* creates for several object types.

4.3 Starting Up EH

The EH takes the root node of a domain AST (i.e. a *GOMT-DomainTheory* object) as its only argument. The input argument can be the root object of the entire domain AST in the case when a new

specification is being created, or it can be the root of a partially finished specification that was saved from a previous session. The main driver function is named *EH* and is designed as a *while loop* that executes until the user chooses the "EXIT Elicitor-Harvester" options from the main menu. Before entering the loop, the *Init-Spec-Tree*, *Fill-Data-Dict*, and *Print-Welcome-Message* functions are called. *Init-Spec-Tree* creates a class called *Op-Library* that is used to store pure functions[5].

```
*** Welcome to the Elicitor-Harvester! ***

The spec is: #4<Missile - a GOMT-DOMAINTHEORY>
Your Working Specification is: Missile

You will be asked a series of questions about your specification
Type in your responses giving names of domain objects if possible.
If you're not sure of the domain name, enter one you think is close.


MAIN MENU

(0) None of these
(1) EXIT Elicitor-Harvester
(2) Specify System INPUTS
(3) Specify System OUTPUTS
(4) Specify INTERNAL UPDATE functions of System
(5) Perform CLEAN-UP Functions
(6) SAVE Sub-Menu
(7) PRINT Specification

Choose the Specification function you want to perform =>
```

Figure 14 The start up message and Main Menu

When the EH tool starts up with a new specification, the initialization rules named *init-eh-ruleset* perform several tasks on the domain objects. The *eh-used* and *eh-pred-used* attributes are reset; the *connection-to-class* maps are set for *GOMT-aggregate-class* objects; and duplicate global data types[6] are purged from the AST. The *Fill-Data-Dict* function will execute only for a new specification that doesn't have a data dictionary defined yet.

[5] Pure functions are operations that simply act on the input arguments and return some value. Pure functions differ from domain operations in that they don't act on domain attributes. Pure functions are often mathematical functions such as SquareRoot(x), which takes a number as its input argument and returns the square root of the number.

[6] Global data types are stored in the dom-global-types map as DomTypeObj objects. At the time of this research, all data types loaded through the Z parser were put in the global area. Therefore, data types declared in more than one class would end up as duplicates in the domain.

The interface starts by showing the main menu as shown in Figure 14. Choices (2), (3), and (4) take the user to the heart of the EH functionality where domain items can be created, modified, and selected for the specification. Choice (5) helps the user "clean up the specification" by moving constants and data types from the global level to a private level within a class if desired. Choice (6) brings up the sub menu of save options, and Choice (7) prints the objects currently selected for use in the specification.

4.4 Specifying Domain Items

Options (2), (3), and (4) are used for specifying the required parts of an application.

- Option (2) allows the user to identify the domain items needed to accommodate new data that is to be input to the application. Data is usually put into an application through input operations like *set-attribute* functions. Option (2) helps the user choose which operations can accept input data and which attributes are required to store the new data. For example, an input could be the insertion of a new record into a database.

- Option (3) allows the user to identify the domain items needed to output data from the application to an output medium such as a report, a screen display, or another application. Output Data is usually accessed through an operation like a *get-attribute* function, which reads an attribute or set of values from a domain object, possibly performs some processing on the data, then outputs the result in some defined format.

- Option (4) allows the user to identify the domain items needed to support internal updates to the application. Internal updates refer to changes in the state of the system and usually occur when an event causes the value of some attribute to be changed. Internal updates do not directly result from input and do not directly cause an output. However, internal updates are often indirectly related to inputs and outputs. Consider an application being specified in the SCHOOL domain, for example. The input to the application may be the insertion of a new student record. The insertion event causes an internal update to an attribute called *total_enrolled_students*. This update may in turn trigger a function that outputs the number of male versus female students now enrolled.

When choice (2), (3), or (4) is selected from the main menu, the function *Process-Specs* is called. *Process-Specs* has one argument of type string called *stage*, which indicates whether the user wants to specify an input, output, or some internal update function. *Process-specs* is designed to loop until the user wants to return to the main menu. Even though the user may be defining specifications for the input, output, or internals stage, the actual processing is the same in all cases. During prototyping it was observed that there were very

few differences in the way items were specified during different stages, so the decision was made to combine the processing into one common function. Future study may show it to be more beneficial to separate the functionality, but for this version of EH, the distinction between stages is only in the user's mind and in the stage variable that remains for future enhancements. Depending on which choice the user picks from the main menu, one of the three prompts shown in Figure 15 will be displayed.

```
Enter one of your desired INPUTS or return key to return to Main Menu
Enter one of your desired OUTPUTS or return key to return to Main Menu
Enter one of your desired INTERNALS or return key to return to Main Menu
```

Figure 15 Screen Display: User prompts for the name of an input, output, or internal update.

At this point the user can enter the name of a domain item that needs to be included in the specification. For example, suppose the user is using the CRUISE MISSILE domain and wants the level of the fuel tank as an output. The input from the keyboard, shown in **bold,** and the resulting matches from the data dictionary are shown in Figure 16. In this case, only one match was found for *level*, namely the *fuel_level* attribute in the *FuelTank* class, and that is the one the user wanted. When the user enters (1) to choose the correct object, the EH asks the user to indicate the action to be performed on that object. Figure 16 shows the actions the user can take for a chosen object. The actions include:

1. REJECT - allows the user to back up to the previous prompt if the object was selected in error. No changes occur in the specification.

2. SELECT - the user can "select" the object for use in the specification. Section 4.4.1 describes the processing that takes place when items are selected for inclusion in the specification.

3. ADD - the user can decide he wants to create a new object instead of the one shown. Section 4.5 explains how new items can be created and added to the specification AST.

4. MODIFY - the user can make changes to the object, such as renaming it or redefining the data type. Section 4.4.2 covers the methods used for modifying specification items.

```
Enter one of your desired INPUTS or return key to return to Main Menu
level

(0) None of these
(1) ATTRIBUTE: FuelTank.fuel_level : Real

Enter the number of an object you will want to to use for 'level'=> 1
Object selected

What is your preferred action on the object:
  ATTRIBUTE: FuelTank.fuel_level : Real

(1) REJECT: Do not want this object
(2) SELECT: Include this item in the Spec (you will have a chance to modify it)
(3) ADD:    I want to create a new object
(4) MODIFY: I want to change this object
Enter your choice of action => :
```

Figure 16 Screen Display: Choosing objects and the action options.

4.4.1 Selecting Objects for the Specification

When the user chooses to select an object for the specification, the object is passed to the *Perform-Select-Actions* function. This function calls *Full-Obj-Description*, which generates a pretty print description of the object to be selected. The *Full-Obj-Description* function is similar to the *Obj-Description* function described earlier, but produces a more complete object description for some objects. The user is given a chance to check the object for errors or possible modifications before finally selecting it, as shown in Figure 17.

```
ATTRIBUTE:  fuel_level : Real

(0) None of these
(1) SELECT to use 'as is' in the Specification
(2) MODIFY it before SELECTing for the Specification
(3) DO NOT SELECT for use in the Specification

What do you want to do with this object? =>
```

Figure 17 Screen Display: Choosing to select an object or modify first.

If the object is acceptable as is, the user chooses (1) and the function *Select-Spec-Items* is called to set the *eh-used* attribute to true for the selected object. In addition to the object selected by the user, the EH automatically selects all required "supporting objects". Supporting objects are other objects needed in the specification to completely define the selected object. For example, if an attribute is selected, then the data type of the attribute should also exist in the specification. The EH decides which other objects are necessary depending on the type of object being selected. The following rules of thumb are applied when selecting supporting objects:

- If the selected object is a class (a GOMT-Class[7] in the domain AST):

    1. Select the ancestor classes through inheritance. Selecting these super-classes (or parent classes) from which a class is inherited is consistent with the object-oriented concept that all inherited attributes and methods should be accessible to an instantiated object.

    2. Select the aggregate ancestor classes. Selecting the aggregation parent class ensures that if a part of an aggregate system is selected, then the framework that houses that part is also placed in the specification. These objects are gathered by using the *ancestors-of-class* function.

- If the selected object is not a class:

    1. Select all objects contained in the subtree of the selected object including predicate objects. These objects are gathered by using the *descendants-of-class* function.

    2. Select the object that is the aggregation parent object in the domain AST (named as the *parent-expr* attribute) of the selected object. For example, the parent of the *Parameter* object of Figure 18 is the operation object identified as a *GOMT-Op* object.

    3. Select the aggregate ancestor classes. For example, if the an operation in the *FuelTank* class of the CRUISE MISSILE domain shown in Figure 10 was selected, the *PropulsionSystem* class and the *Missile* class would be the ancestor class objects selected as supporting objects.

- If the selected object is a parameter, select the predicates that belong to the same operation as the parameter selected. These predicates are the pre-conditions and post-conditions of the operation and will most likely include the parameter as one of its identifiers.

- If the selected object or one of its supporting objects has a data type associated with it (the *has-atype* map is defined), select the data type object (a *DomTypeObj* in the domain AST) as a supporting object. Since data types are not defined as tree attributes, they are not part of the subtree and thus do not get selected in the preceding steps.

- If the selected object or one of its supporting objects has a class mapped to it through the *has-aclass* map, the *connection-to-class* map, or the has-associative-object map, then select the class object as a supporting

[7] A GOMT-Class is an object class defined in the DOM AST that represents a class in the object-orented sense. A class is a self-contained structure that represents some real-world object. The class contains definitions of attributes that describe the features of the class, operations describing the actions that can be performed on the class, the valid states of the class, and the events that can cause state changes in the class.

object. This case can occur when a class attribute or an operation parameter is defined as a class or a set of classes, such as *fac* : *P Faculty*.

- If a predicate is selected as a supporting object:

  1. Mark the predicate for the specification by setting the *eh-pred-used* map to True.

  2. Find all domain objects represented as identifiers in the predicate. This step is accomplished by calling the *Map-ID-to-Obj* function described in 4

  3. For each represented object, make a recursive call to the *Select-Spec-Items* function to mark all its supporting objects for the specification.

For an example of selecting objects, refer to the attribute *fuel_level* shown in Figure 17. Besides *fuel_level*, other objects selected include *FuelTank*, the class that contains *fuel_level*; *JetPropSys* and *Missile*, the aggregate ancestors of *FuelTank*; and *Real*, the *DomTypeObj* that is the data type of *fuel_level*.

Generally, when operations are selected to the specification, many supporting objects can also be selected automatically. The operation subtree is fairly complex, as shown in Figure 18, and several parts of the subtree are associated with other parts of the AST. For example, a predicate will usually contain identifiers that are stored as classes, class attributes, or constants elsewhere in the AST. Those identifiers as well as parameters have data types associated with them that are also stored elsewhere. The method for finding the objects represented by the predicate identifiers is explained in Section 4.4.1.1. Therefore, by helping the user to select the operations required for the application being specified, the EH can identify several parts of the domain tree needed for the specification.

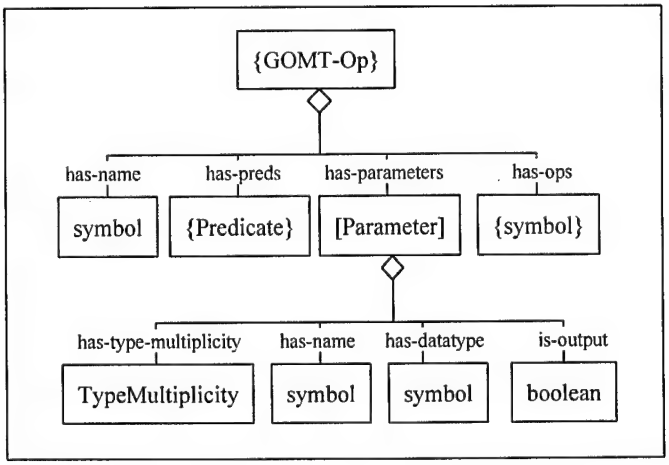

Figure 18 Structure of the Operation Subtree

4.4.1.1 Mapping Predicates to Domain Objects

If an operation is selected for the specification, the predicates of the operation are also selected, since they are part of the subtree, as shown in Figure 18. Predicates selected for the specification provide a rich opportunity to identify many other domain objects represented by the predicate identifiers that should also be included in the specification. Consider an operation called *CalcPropWt*, shown in Figure 19, which is defined in

```
OPERATION:  CalcPropWt
    OUTPUT PARAMETER: prop_wt  IS: Real
    PREDICATE:  prop_wt! = fueltank.tank_weight + jetengine.engine_weight
```

Figure 19 CalcPropWt: an operation in the CRUISE MISSILE domain

the *Propulsion System* aggregate class in the CRUISE MISSILE domain. Since the predicate contains identifiers that refer to the classes *FuelTank* and *JetEngine* and the class attributes *tank_weight* and *engine_weight*, those objects are also selected for the specification by recursively calling the *Select-Spec-Items* function for each of the predicate objects.

The difficulty in finding the domain objects corresponding to the predicate identifiers lies in the fact that the *IDname* objects, which store predicate identifiers, do not currently have maps defined to the domain objects that they represent. Since predicates hold so much domain knowledge, it was important to create a function called *Map-ID-to-Obj* that maps *IDname* objects to the corresponding domain objects. There are many subtle problems in making the *IDname* to *Obj-Object* mapping.

- There is currently no automated type checking performed on predicate identifiers when they are parsed in. Therefore, the names are not guaranteed to be spelled correctly.
- There can also be attributes in different classes with the same name.
- A predicate can use bound variables like s and c in the predicate:
  $ms = \#\{s : Student; c : GradClass \mid s \in advised? \wedge (s,c) \in member\_of \wedge c.program \neq DS\}$
  that represent domain objects but whose names could not be directly matched to domain names.

When mapping identifiers from a complex predicate like the one in the third bullet above, it is desirable to identify the context of each variable so that all variables, including bound variables, can be accounted for and mapped to the proper domain objects. Bound variables can be used in three types of expression: basic

declarations (*s:Student*), element expression (*s ∈ advised?*), and component expressions (*c.program*). These expression types are represented as predicate subtrees in the Unified-Object model. The bound variables *s* and *c* are stored in the IDname objects in the left branch of the subtrees shown in Figure 20.

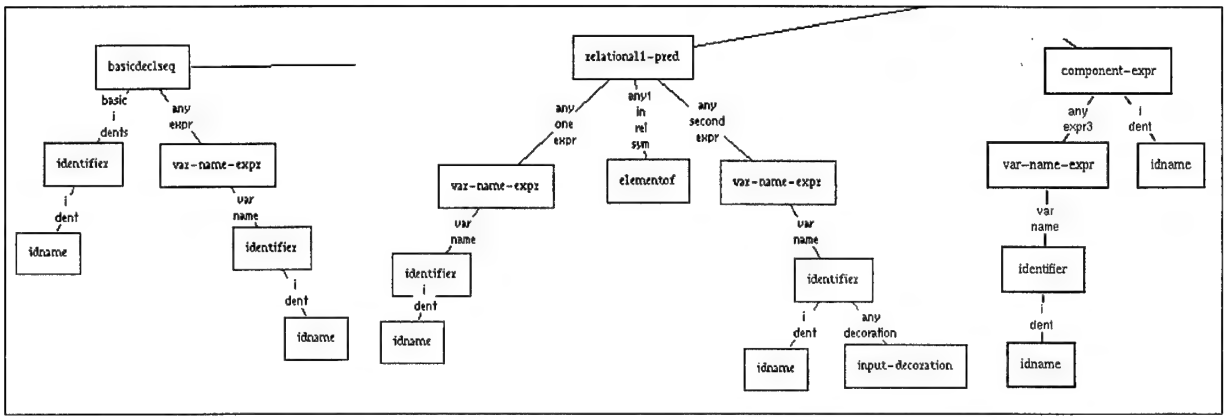

Figure 20 Predicate AST in Unified-Object Model

When the EH gathers the *IDname* objects from the predicate AST, it has no way of knowing which identifiers are bound variables or which objects are represented, so several maps were created, as shown in Figure 21, to aid in the mapping process. The *Map-ID-to-Obj* function first sets the *ID-wrd* map by using the data dictionary matching rules to find domain objects that perfectly match the names in the predicate.

```
var id-obj             : map(IDname, Obj-Object) = {||}
var ID-wrd             : map(IDname, Aword) = {||}
var is-BasicDecl-of    : map(IDname, IDname) = {||}
var is-element-of      : map(IDname, IDname) = {||}
var has-component-var  : map(IDname, IDname) = {||}
var pred-mapped        : map(Predicate, boolean) = {||}
```

Figure 21 Maps for processing predicates

The function then searches the predicate AST for instances of *BasicDeclSeq*, *ElementOf*, and *Component-Expr* nodes and sets the *is-BasicDecl-of*, is-*element-of*, and *has-component-var* maps so that each bound variable identifier is mapped to a name that should be identifiable in the domain. The algorithm to perform the mappings checks the ancestor nodes of each IDname for a *BasicDeclSeq*, *ElementOf*, and *Component-Expr* node. If one is found, the IDname is mapped to the other IDname identifier in the subtree. In the predicate AST of Figure 20 the IDname storing the first *s* would be mapped to the *Student* IDname through the *is-BasicDecl-of* map and the *Student* IDname would be mapped to the *Student* object in the domain through the *id-*

obj map. If any identifier, such as *s*, is found with an *is-BasicDecl-of* map defined, then all other identifiers in the predicate with the same name are mapped to the same domain object, i.e. the *Student* object.

If there are no *BasicDeclSeq* nodes, but an *ElementOf* node is found, the bound variable will represent an element in some set of objects. In this case, the bound variable will be mapped to the same *GOMT-Class* object pointed at by the *id-obj* map of the associated IDname. For example, if the declaration `s:Student` was not present in the predicate, the expression `s ∈ advised` would cause the *advised* IDname to be mapped through the *id-obj* map to the input parameter *advised*, which is declared as a set of *Student*. Then *s* would be mapped to the *has-aclass* attribute that identifies the *GOMT-class*, which is the type of the *advised* parameter. In the end, *s* is mapped to the *Student* object through *s*'s *id-obj* map.

Bound variables found in component expressions are assumed to have been previously identified in a basic declaration or element-of expression, and therefore should have already been mapped. In the case where a predicate contains ambiguous identifiers, they should be prefixed via dot notation with the GOMT-class name, a connection name, or a bound variable. If the ambiguous identifier is prefixed with a bound variable, the algorithm sets the ambiguous variable's id-obj to the class attribute of the same name that is contained in the GOMT-class mapped to the bound variable. If the algorithm fails to map all ambiguous variables in the predicate, the user is finally asked to enter the proper object.

4.4.2 Modifying Objects

There will be times when the application engineer wants to change some item while defining the specification. The application engineer may want to tighten constraints or complete constant or data types definitions that were left incomplete in the domain model. In most cases, before a domain item is selected for use in the specification, a description of the item is displayed and the user is given the chance to modify the item. Of course the ability to modify given parts of the domain can be restricted by the knowledge base administrator based on the requirements of Chapter 3.

The menus shown in Figure 16 and Figure 17 have an option for making modifications to specification items. Before finally selecting items for use in a specification, the user is given the opportunity to make changes if necessary. The allowable changes are defined in Chapter 3. In the *fuel_level* example of Figure 17, if the selected object is not specified the way the application engineer wants it, he has the choice to modify the object before adding it to the specification AST. If the *modify* option is chosen, the object is passed to the

Perform-Modify-Actions function. The function first creates a database instance for the modification rules to use. The database is defined by the EH-Object of Figure 22 and the subtype called *Mod-Object* shown in Figure 23.

```
% superclass of Add-object and Mod-object
var EH-Object      : object-class subtype-of GOMT-Aggregate-Class
var has-obj-object : map(EH-Object, Obj-Object) = {||}
var has-stage      : map(EH-Object, string) = {||}
var has-idnames    : map(EH-Object, set(IdName))
                     computed-using has-idnames(x) = {}
```

Figure 22 EH-Object database declaration

The object to be modified is pointed to by the *has-obj-object* map, and the *Mod-Object* database is passed to the modification rules, called *mod-obj-rules*, by the forward reasoning *preorder-transform* function. The first time the *Mod-Object* passes through the rules, the set of rules check the attributes of the object under modification that are allowed to be changed and append an appropriate string to the *options* map sequence.

```
var Mod-Object                 : object-class subtype-of EH-Object
var prev-name                  : map(Mod-Object, symbol) = {||}
var options                    : map(Mod-Object, seq(string))
                                 computed-using options(x) = []
var chosen-option              : map(Mod-Object, string) = {||}
var List-mod-name?             : map(Mod-Object, boolean) = {||}
var List-Mod-Avalue?           : map(Mod-Object, boolean) = {||}
var List-Mod-Datatype?         : map(Mod-Object, boolean) = {||}
var List-Mod-ClassType?        : map(Mod-Object, boolean) = {||}
var List-Mod-TypeMult?         : map(Mod-Object, boolean) = {||}
var List-Mod-Param?            : map(Mod-Object, boolean) = {||}
var List-Mod-Pred?             : map(Mod-Object, boolean) = {||}
var List-Mod-class?            : map(Mod-Object, boolean) = {||}
var List-Mod-TypeMultiplicity?: map(Mod-Object, boolean) = {||}
var List-Mod-Attr?             : map(Mod-Object, boolean) = {||}
var List-Mod-Connection?       : map(Mod-Object, boolean) = {||}
var List-Mod-Connection-Class?: map(Mod-Object, boolean) = {||}
var List-Mod-Connection-Mult? : map(Mod-Object, boolean) = {||}
var List-Mod-Assoc?            : map(Mod-Object, boolean) = {||}
var List-Mod-Constant?         : map(Mod-Object, boolean) = {||}
var List-Mod-Operation?        : map(Mod-Object, boolean) = {||}

var List-Add-Operation?        : map(Mod-Object, boolean) = {||}
var List-Add-Datatype?         : map(Mod-Object, boolean) = {||}
var List-Add-Param?            : map(Mod-Object, boolean) = {||}
var List-Add-Pred?             : map(Mod-Object, boolean) = {||}
var List-Add-Attr?             : map(Mod-Object, boolean) = {||}
var List-Add-Connection?       : map(Mod-Object, boolean) = {||}
var List-Add-Assoc?            : map(Mod-Object, boolean) = {||}

var List-Del-Operation?        : map(Mod-Object, boolean) = {||}
var List-Del-Param?            : map(Mod-Object, boolean) = {||}
var List-Del-Pred?             : map(Mod-Object, boolean) = {||}

var mod-done?                  : map(Mod-Object, boolean) = {||}
% flag for data-dictionary update when name changed
var mod-name-done?             : map(Mod-Object, boolean) = {||}
```

Figure 23 Mod-Object database declaration

Each rule that adds a string to the options map sets the corresponding boolean map (for example, *list-mod-param*?) to true, which prevents that rule from firing again. The list of strings held in *options* is passed to the

```
OPERATION:  CalcPropWt
    OUTPUT PARAMETER: prop_wt  IS: Real
    PREDICATE:  prop_wt! = fueltank.wieght + jetengine.engine_weight

(0) None of these
(1) Change Name
(2) Modify Parameter
(3) Modify Predicate
(4) Add Parameter
(5) Add Predicate
(6) Delete Parameter
(7) Delete Predicate

How would you like to modify 'CalcPropWt'? =>
```

Figure 24 Screen Display: Modification options list

Print-String-List to be displayed for the user. Figure 24 shows a screen display of the possible modifications that can be made to the operation *CalcPropWt*, and the prompt asking for the user to input a choice.

The string corresponding to the number chosen by the user (e.g. "Modify Parameter") is set as the value in the *chosen-option* map of *Mod-Object*. If the user choice was not (0), the *preorder-transform* function is called again to check the *mod-obj-rules*. The rule looking for the string held in chosen-option will fire and call a function to perform the appropriate actions for the option chosen to be modified. For example, if the user chooses to "Modify Parameter", the rule *Modify-Parameter-rule* would fire causing the *Modify-Parameter* function to execute. If the option chosen relates to a domain object that has a subtree, i.e. is not a leaf node, such as a parameter, the function will recursively call *Perform-Modify-Actions* so the user can pick the part of the chosen option to modify. If the chosen option does relate to a leaf node, such as the name or value of a class attribute like *fuel_level*, the appropriate function will control the user interface to request needed information from the user. Some cases may be as simple as printing out

`Enter the new name for 'fuel_level' =>`

and reading in the string, but other cases may lead into much more involved interactions.

Consider an example where the application engineer wants to change the data type of *fuel_level* from type *Real* to a more specific data type called FUEL_LEVEL_TYPE that has enumerated values. Figure 25 shows a series of interactions between the EH and the user. After the prompt `How would you like to modify 'fuel_level'? =>`, the *Modify-Datatype* function takes control and searches for predicates that

contain *fuel_level* as an identifier. The function gathers the set of all data types of the named identifiers in the predicates and prints out that list of data types for the user to view. The reason for this first subset of data types is to minimize the amount of information given to the user by some intelligent heuristic. Predicates often contain identifiers of similar data type, especially if the predicate defines a mathematical calculation. If the first list of data types doesn't contain the desired choice, the entire list of domain data types is printed out. This list could be quite long in a large domain; thus the use of better heuristics in these situations is desirable and an open area for further study.

```
What is your preferred action on the object:
  ATTRIBUTE: FuelTank.fuel_level : Real

(1) REJECT: Do not want this object
(2) SELECT: Include this item in the Spec (you will have a chance to modify it)
(3) ADD:    I want to create a new object
(4) MODIFY: I want to change this object
Enter your choice of action => :2

ATTRIBUTE:  fuel_level : Real

(0) None of these
(1) SELECT to use 'as is' in the Specification
(2) MODIFY it before SELECTing for the Specification
(3) DO NOT SELECT for use in the Specification

What do you want to do with this object? =>  2

ATTRIBUTE:  fuel_level : Real

(0) None of these
(1) Change Name
(2) Modify Datatype

How would you like to modify 'fuel_level'? =>  2

(0) None of these
(1) DATATYPE: Real, VALUES:

Choose the new datatype for 'fuel_level: Real' =>  0

(0) None of these
(1) DATATYPE: Boolean, VALUES:
(2) DATATYPE: Digit, VALUES:
(3) DATATYPE: Char, VALUES:
(4) DATATYPE: Integer, VALUES:
(5) DATATYPE: Nat_1, VALUES:
(6) DATATYPE: Nat, VALUES:
(7) DATATYPE: AF_MODELS, VALUES:
(8) DATATYPE: KILOMETER, VALUES:
(9) DATATYPE: KPH, VALUES:
(10) DATATYPE: RADIAN1, VALUES:
(11) DATATYPE: RADIAN2, VALUES:
(12) DATATYPE: DEGREE, VALUES:
(13) DATATYPE: SIMTIME, VALUES:
(14) DATATYPE: MODEL_TYPE, VALUES:
(15) DATATYPE: SEQ_Char, VALUES:

Choose the new datatype for 'fuel_level: Real' =>  0

Must Create type for 'fuel_level'
```

Figure 25 Screen Display: Modifying the data type of an attribute

In this example, the user still doesn't see the data type he wants, so he chooses (0) again.

At this point, the EH knows the data type does not exist and must be created. A new database called *Add-Object* is initialized and passed to the function *Perform-Add-Actions*, which is responsible for creating new specification items and adding them to the AST. The method for creating new objects is discussed in the following sections.

After the second pass through the *mod-obj-rules*, the *mod-name-done?* map of the *Mod-Object* database is checked to see whether the object name was changed. If so, the function *Update-Data-Element-Name* is called to update the data dictionary with the new name. Finally, the new object description is printed out by calling the *Full-Obj-Description* function; and the modified object is passed from the *Perform-Modify-Actions* function.

4.5 Adding New Objects

There are many cases when an application engineer may want to create new items for the specification that were not defined in the domain. A new application may require operations to be defined at a lower level than were needed in the domain. It may be beneficial to create more specific data types to define constraints on certain data. New constants may make the specification more clear and understandable. Associations between objects may make sense for specific applications, but not for the overall domain. After new items are created, they must be grafted onto the existing specification AST in the correct location. This section describes the techniques used for creating new specification items and adding them to the specification tree.

4.5.1 Creating Objects Using Backward Reasoning

Adding items to the specification tree is a good situation for using backward reasoning, because very little is known about the new item to be created. A great deal of information already exists in the domain tree that can be used to infer much of the data needed when adding new items to the specification. The process of adding a new operation to a specification can be used as a good example of how backward reasoning can be applied. All that is initially known about a new operation is the name of an input or output, which the user has typed in. The input and output parameters, data types, pre-conditions, post-conditions, and name are needed to completely define the new operation.

The EH guides the user through the process of defining the operation by first asking for the post-conditions (predicates) that define the output of the operation (most operations will have a single output, which

essentially makes them a function). Once a predicate is parsed, the EH can search through the specification AST for information relating to the predicate. For example, the EH may search for objects whose names match the identifiers in the predicate, data types or class types of those identifiers, classes to which the identifiers belong, and identifiers that may be candidates for input parameters. Of course the user must make some choices and validate the choices made by the EH, but by using backward chaining, the EH restricts the number of choices to a manageable level for the user and guides the user through the process one step at a time.

The backward reasoning process requires three main components, which are discussed in the following sections: a database to store information discovered during the process, a set of rules that access and update the database, and a reasoning engine that controls the execution of the rules.

4.5.2 The Backward Chaining Rule Base

The rules are implemented as REFINE rule constructs. The rules used for the backward chaining are designed so that each rule only solves a small piece of the problem. The rules are designed to use the recursive nature of the backward chaining algorithm to incrementally add information to the database in a certain order so as to get the most use out of each piece of data acquired. The backward reasoning engine works its way through applicable rules to achieve a goal. Figure 26 shows the REFINE code of a rule and the AST structure used to store the rule. The engine tries to fire rules that have the current goal in the consequent by making all the

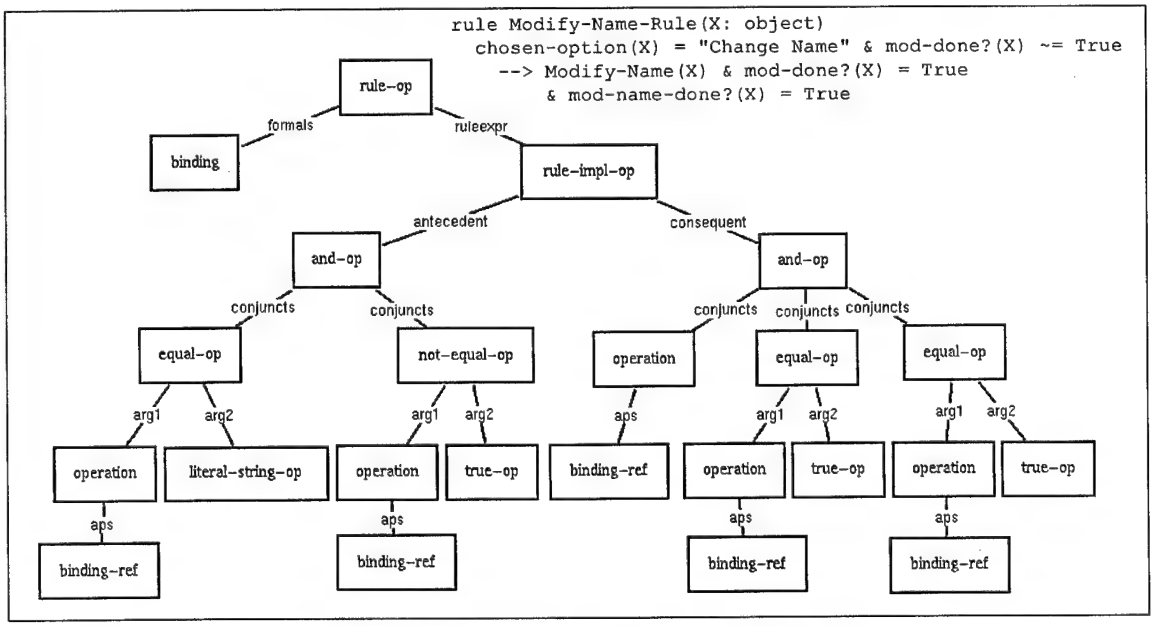

Figure 26 AST created by REFINE to store a rule

premises true. For the rule in the figure to be chosen as a candidate rule, the goal would be to find a value for *mod-done?*. Once the applicable rules are found, the engine checks the premises to see if they are defined by the database.

In Figure 26, the premises are `chosen-option(X) = "Change Name"` and `mod-done?(X) ~= True` and are stored as the *equal-op* and *not-equal-op* subtrees below the *antecedent* map in the AST. The database (*Mod-Object* in this example) defines the maps *chosen-option* and *mod-done?*, which map to a literal string and a boolean operator values respectively. If one of these maps is undefined (not yet given a value), the engine makes the undefined premise a subgoal and recurses to look for other rules that may be able to derive that subgoal and so on until a rule is found where all premises are satisfied. For a premise to be satisfied the corresponding database map must have a value defined and the value must cause the premise to be true. In this example, *chosen-value* must map to the string "Change Name" and *mod-done?* must map to false to make both premises evaluate to true. When all premises of a rule are satisfied, the rule can execute the consequent portion of the rule, which usually includes a call to a function that performs some type of database update action. In this example, the rule would call the function *Modify-Name*. These functions use information harvested from the specification AST and user inputs to make updates to the database. The updates to the database often cause premises of other rules to be satisfied. When a rule fires, the engine returns to the previous unfinished rule, as expected from a recursive function, to see if any database updates have satisfied its premises.

4.5.2.1 The Backward Chaining Database

The database used by the backward reasoning engine, a subtype of *EH-Object* called *Add-Object*, is shown in Figure 27. The state of the database changes after each rule execution until it contains all data needed to complete the final goal or return a message indicating failure to achieve the goal. The goal of the backward reasoning process is to create a new specification item such as an operation or data type. The engine usually works through several subgoals in the process of solving the main goal. For example, before creating a new operation, the pre-conditions, post-conditions, input and output parameters, and operation name must all be created. Many of these subgoals are broken down into still smaller subgoals. Each time a rule is fired and a subgoal is achieved, a map in the database is given a value.

For example, a subgoal may be to have a predicate defined for the post-condition of an operation. A goal called *predicate-retrieved?* matches the consequent in the rule *Retrieve-Predicate-Rule* and since all the

premises of *Retrieve-Predicate-Rule* are satisfied, the rule fires. When the rule fires, it calls the function *Retrieve-Predicate*, which parses in the predicate string typed in by the user. The new predicate is placed into the *has-apred* map of the database; and the *predicate-retrieved?* map is set to true. As the subgoals are

```
var Add-Object                  : object-class subtype-of EH-Object
var add-obj-name                : map(Add-Object, string) = {||}
var has-stage                   : map(Add-Object, string) = {||}
var is-class-type               : map(Add-object, symbol) = {||}

var new-item-names              : map(Add-object, set(string))
                                   computed-using new-item-names(x) = {}
var has-postcond                : map(Add-Object, Predicate) = {||}
var has-preconds                : map(Add-Object, set(Predicate))
                                  computed-using has-preconds(x) = {}
var has-inparams                : map(Add-Object, set(Parameter))
                                  computed-using has-inparams(x) = {}
var has-outparam                : map(Add-Object, Parameter) = {||}
var has-apred                   : map(Add-Object, Predicate) = {||}
var has-set-of-types            : map(Add-Object, set(DomTypeObj))
                                  computed-using has-set-of-types(x) = {}
var has-set-of-classes          : map(Add-Object, set(GOMT-Class))
                                  computed-using has-set-of-classes(x) = {}
var var-list                    : map(Add-object, seq(string))
                                  computed-using var-list(x) = []
var outparam-name               : map(Add-Object, symbol)  = {||}
var inparam-names               : map(Add-object, set(symbol))
                                  computed-using inparam-names(x) = {}
var has-goal                    : map(Add-Object, symbol)  = {||}
var goal-list                   : map(Add-object, seq(symbol))
                                  computed-using goal-list(x) = []
var aword-seq                   : map(Add-Object, seq(Aword))
                                  computed-using aword-seq(x) = []
var derived-from                : map(Add-Object, set(Obj-Object))
                                  computed-using derived-from(x) = {}
var DomConstant-done?           : map(Add-Object, boolean) = {||}
var predicate-retrieved?        : map(Add-Object, boolean) = {||}
var idnames-retrieved?          : map(Add-Object, boolean) = {||}
var derived-types-retrieved?    : map(Add-Object, boolean) = {||}
var derived-classes-retrieved?: map(Add-Object, boolean) = {||}
var outparam-name-retrieved?    : map(Add-Object, boolean) = {||}
var unmatched-vars-retrieved?   : map(Add-Object, boolean) = {||}
var pred-vars-identified?       : map(Add-Object, boolean) = {||}
var Inparams-chosen?            : map(Add-Object, boolean) = {||}
var Inparams-done?              : map(Add-Object, boolean) = {||}
var aword-seq-done?             : map(Add-Object, boolean) = {||}
var derived-from-done?          : map(Add-Object, boolean) = {||}
var type-chosen?                : map(Add-Object, boolean) = {||}
var postcond-done?              : map(Add-Object, boolean) = {||}
var precond-done?               : map(Add-Object, boolean) = {||}
var Predicate-done?             : map(Add-Object, boolean) = {||}
var GOMT-Op-done?               : map(Add-Object, boolean) = {||}
var Outparam-done?              : map(Add-Object, boolean) = {||}
var Inparam-done?               : map(Add-Object, boolean) = {||}
var has-class-name?             : map(Add-Object, boolean) = {||}
var has-name?                   : map(Add-Object, boolean) = {||}
```

Figure 27 Add-Object database declaration

satisfied, database values are filled in until all values needed for the original goal are present and the new specification item can be added to the AST.

4.5.2.2 Backward Reasoning Algorithm

The backward chaining engine designed was modeled after the standard algorithm found on pages 96-97 of [7]. Since REFINE does not support backward chaining, the algorithm had to be built manually and adapted to the REFINE language. Backward chaining requires the ability to access the premises and values coded into the REFINE rules, which REFINE parses into an AST as shown in Figure 26.

Several functions were written to manipulate the rule subtree to access the values stored in it. The algorithm is shown in Figure 28.

```
GLOBALS: goal list, rule list, database object

make list of top-level goals
loop while goal list not empty
   SATISFY-GOAL()
      set current goal to first goal
      make list of candidate rules for current goal
      goal satisfied = False
      loop while goal not satisfied by rule or candidate rule list empty
         set current candidate to first candidate
         make list of premises of current candidate
         premise satisfied = true
         loop while premise list not empty and premise satisfied
            set current premise to first premise
            if premise parameter defined in database then
               if premise parameter = database value then
                  premise satisfied = True
                  delete first premise from list
               else premise satisfied = False
            else
               if rules exist to derive premise then
                  prepend premise argument to goal list as a subgoal
                  recurse to SATISFY-GOAL for new goal
               else
                  ask user for value of premise parameter
                  add value to the database
         end loop
         if premise satisfied
            fire rule
            goal satisfied = True
         else
            delete first rule from candidate list
      end loop
      empty list of candidate rules
      delete first goal from goal list
   END SATISFY-GOAL
end loop
return database
```

Figure 28 Algorithm for the backward reasoning engine

4.5.3 Examples of Creating Objects

The next two sections take the reader through the process of creating an example operation and data type using the CRUISE MISSILE domain. The process for creating new domain items varies from the approach used for modification. Recall how the modification method created a new database for each object while recursively calling *Perform-Modify-Actions* for each lower level object until a leaf node was reached. Each object was acted upon independent of the parent and sibling objects. When creating a new domain item such as an operation, child objects often need to know information about other child objects, even though they have not been created yet. For instance, a parameter should appear in a predicate, and the predicates need to check parameter data types, creating a sort of circular dependence. Also, the operation cannot be added to the specification tree until the reasoning engine is done creating the entire operation because subtree objects are generally processed from the bottom up. Therefore, the creation process was designed to operate on the entire item as a whole instead of the sub-objects individually. The *Add-Object* database stores data for the child objects temporally until the entire item is ready to be built and added to the specification. Sometimes while processing one item, it is discovered that another item needs to be created to support the current item. For example, an operation predicate may contain a call to another operation. In this case, the current item process is suspended and *Perform-Add-Actions* is called recursively to create the other item and return to the original upon completion.

4.5.3.1 Creating an Operation

Assuming the CRUISE MISSILE domain is loaded and the user has selected choice (3) from the main menu (Specify system outputs), the EH calls the function *Process-Specs*, which prints out the first prompt shown in Figure 29.

When the user enters "Prop_Wt" (propulsion system weight), the *Get-Phrase* function reads in the string, creates an *Aword* structure for Prop_Wt, and accesses the data dictionary through the *Match-Word* function, which makes four matches to domain objects. Control passes to the function *Get-Actions*, which finds out whether the user wants to perform a SELECT, ADD, or MODIFY action on the object chosen. *Get-Actions* calls the function *Choose-Objects*, which prints out the matched objects using the *Print-Obj-List* function. The *JetPropulsionSys* aggregate class does have an attribute called *prop_weight*, but for this example, assume the

```
Enter one of your desired OUTPUTS or return key to return to Main Menu
Prop_Wt

(0) None of these
(1) CLASS: JetPropulsionSys
      HAS-ATTRIBUTES: prop_weight;  prop_fuel;
(2) CONNECTION: Missile.propsys : JetPropulsionSys
(3) ATTRIBUTE: JetPropulsionSys.prop_fuel : Real
(4) ATTRIBUTE: JetPropulsionSys.prop_weight : Real

Enter the number of an object you will want to to use for 'Prop_Wt'=> 0
No object selected would you like to create one? y

(0) None of these
(1) A new Primitive Class
(2) A new Aggregate Class
(3) A new Attribute of an existing class
(4) An output Operation
(5) An input Operation
(6) A new Data Type
(7) A new Constant

What kind of object should 'Prop_Wt' be created as? 4
```

Figure 29 Screen Display: Identifying a new operation

user wants to define an operation to calculate the propulsion weight from other attributes. When the user enters (0), the *Get-Actions* function has no objects to work with, so it asks if the user wants to create one. When the user enters 'y' for yes, a *Selected-Obj* object is created with its *has-action* map set to ADD to indicate the desire to create a new object; and *Selected-Obj* is added to the *has-sel-objs* map of the *Aword* structure. Control then returns to *Process-Specs* where the *has-actions* map is checked. The ADD value causes *Perform-Add-Actions* function to be called, which is the main driving function for adding new specifications. *Perform-Add-Actions* takes as its argument an *Add-Object* initialized by the calling function. The calling function will fill in whatever values are known when creating the *Add-Object* database. In this example only the *add-obj-name* ("Prop_Wt") and the *stage* ("OUTPUT") are initially defined. The EH doesn't know what the user wants to create, so the function *Get-Add-Object-Type* is called which prints out the menu shown in Figure 29 along with a prompt. The user enters '4', which sets the *is-class-type* map to "GOMT-Op" and the has-goal map to `GOMT-Op-done?[8] in the database. The *Add-Object* is then passed to the backward reasoning engine with a function call to *Perform-Backward-Chaining*.

[8] REFINE uses a data type called a symbol for most object names and other identifiers. Symbols are generally case insensitive and are identified by placing a tick mark in front of the name.

```
rule Create-Output-GOMT-Op-Rule(X: object)
  has-goal(X) = 'GOMT-Op-done? & has-stage(X) = "OUTPUT" &
  Postcond-done?(X) = True & Outparam-done?(X) = True &
  Precond-done?(X) = True & Inparams-done?(X) = True
    --> GOMT-Op-done?(X) = True & Create-GOMT-Op(X)
```

Figure 30 Sample rule used in the backward reasoning process

When the reasoning engine searches for rules with a consequent expression matching the goal *`GOMT-Op-done?*, it finds *Create-Output-GOMT-Op-Rule* shown in Figure 30. As the reasoning engine parses through the premises, it finds that the *has-goal* and *has-stage* premises are already satisfied in the database, however, *Postcond-done?*, *Outparam-done?*, *Precond-done?*, and *Inparams-done?* are undefined and become a series of subgoals, which must be satisfied before this rule can execute. Each subgoal recursively chains through several rules, which in essence break down a large task into several smaller manageable subtasks. The first subtask in this example is to parse in a post-condition predicate, which the user types in using Z notation, as shown Figure 31, and place it in the *has-apred* map.

```
Return key to quit this action
Enter a Post Condition for Prop_Wt
=> : prop_wt! = fueltank.fuel_weight + jetengine.engine_weight

(0) None of these
(1) CLASS: FuelTank
      HAS-ATTRIBUTES: tank_sim_time;  input_flow_rate;  output_flow_rate;
        fuel_level;  capacity;  tank_weight;  fuel_density;
        fuel_tank_weight;
(2) CONNECTION: JetPropulsionSys.fueltank : FuelTank
(3) ATTRIBUTE: JetEngine.current_fuel_flow_rate : Real
(4) ATTRIBUTE: JetEngine.engine_weight : Real
(5) ATTRIBUTE: FuelTank.fuel_tank_weight : Real
(6) ATTRIBUTE: FuelTank.fuel_density : Real
(7) ATTRIBUTE: FuelTank.tank_weight : Real
(8) ATTRIBUTE: FuelTank.fuel_level : FUEL_LEVEL_TYPE
(9) ATTRIBUTE: JetPropulsionSys.prop_weight : Real
(10) ATTRIBUTE: Airframe.attached_weight : Real
(11) ATTRIBUTE: Airframe.airframe_weight : Real
(12) OPERATION: CalcTotalWeight
      HAS-PARAMETERS: fuel_weight;  fuel_tank_weight;
(13) OPERATION: DetermineFuelWeight
      HAS-PARAMETERS: fuel_weight;

Enter the number of an object you will want to to use for 'fuel_weight'=> 5
```

Figure 31 Screen Display: Defining a post-condition

Several subtasks then process the post-condition as follows:

1. Place the set of IDnames found in the predicate into the *has-idnames* map.

2. Check predicate variables for the decorations ', ?, !, which indicate a final variable[9], an input parameter, and an output parameter respectively, then set the maps *finalparam-names*, *inparam-names*, and *outparam-name* accordingly.

3. Identify the output parameter in the post-condition if not already found in previous step.

4. Create a sequence of *Aword* structures, one for each IDname in the predicate, to help search the data dictionary for matching objects. For each IDname, have the user identify the proper domain object to match the predicate identifier. The spelling of the predicate identifier will be changed to that of the matched domain object. As the screen display of Figure 31 shows, the user selected option (5), which means the identifier originally typed in as *fuel_weight* will be changed to *fuel_tank_weight* in the predicate

5. Store the names of all predicate identifiers not matched to a domain object in step #4. These identifiers may be input parameters or the name of some other object that needs to be created.

6. Print out the list of unmatched predicate identifiers and prompt the user to identify those that are input parameters. For an example, assume the user thought a function called *CalcTankWeight* calculated the current weight of the fuel tank, but this function is not in the domain. The screen display in Figure 32 shows the user rejecting *CalcTankWeight* as an input parameter.

7. For each input parameter identified, ask the user to choose the data type and multiplicity (single, set or sequence), create a *Parameter* object, and store it in the database.

8. Print out the list of unmatched predicate identifiers and ask the user if they are objects that need to be created as shown in Figure 32. For each identifier chosen, initialize a new *Add-Object* database and pass it to the function *Perform-Add-Actions* to create the new item.

9. Gather all predicate variables that have been matched to domain objects into a set.

10. Gather the data types from the set formed in step #9 into a set of types.

11. Gather into a set of *GOMT-Classes* the data types that are classes of the set from step #9.

12. Add the predicate in the *has-apred* map to the set of post-conditions in the *has-postconds* map. Ask the user if there are more post-conditions. If "yes", then reset the *has-goal* to *`Postcond-done?* and the other maps for the post-condition rules to undefined, which will cause the post-condition rules to repeat for the next post-condition.

[9] A predicate identifier marked with a final decoration indicates a class attribute that is changed by the function as a side effect.

13. Ask the user to choose the data type and multiplicity for the output parameter.

14. Create *Parameter* object for the output parameter and place it in the *has-outparams* map.

Next, the pre-conditions are processed in generally the same way as post-conditions except the output parameter is not dealt with. New input parameters identified during the pre-condition processing are handled as in step #7. Finally the *Create-GOMT-Op-Rule* premises are all satisfied and the *Create-GOMT-Op* function is called, which gets the operation name from the user, creates a *GOMT-Op* object and places it in the *has-obj-object* map of the database.

```
(0) None of these
(1) CalcTankWeight

Choose the number of an identifier that should be an input parameter =>  0

(0) None of these
(1) CalcTankWeight

Do any of these unidentified predicate variables need to be created? =>  1
```

Figure 32 Screen Display: Handling unidentified predicate variables

4.5.3.2 Creating a Data Type

The *DomTypeObj* is another item in the domain model that has a fairly complex subtree. A new data type can be created as a new base type or derived from an existing base type by restricting its range with some type of constraint. The call to create an item can come during a modify process as seen in Figure 25. Figure 33 continues the example at the point where the *Modify-Datatype* function calls *Perform-Add-Actions* to create the new data type. The following sub-tasks are performed by the backward reasoning engine and the applicable rules:

1. Ask the user for the name of the new data type.

2. Display a list of domain data types. If the user chooses (0) the rule for creating a *DomBaseType* object will be accessed; if the user chooses one of the existing data types, the rule for creating a *DomDerType* will guide the user through creating a new derived type.

3. In this example, the user chooses to create a new base type and is asked if it will be an enumerated type. The user enters 'y' for yes. If a derived type was to be created, the user would be asked to input the constraint predicate and would be asked if the data type was to be a set or sequence.

```
Must Create type for 'fuel_level'

Enter the name of the new type: FUEL_LEVEL_TYPE

(0) None of these
(1) DATATYPE: SEQ_Char, VALUES:
(2) DATATYPE: Real, VALUES:
(3) DATATYPE: Boolean, VALUES:
(4) DATATYPE: Digit, VALUES:
(5) DATATYPE: Char, VALUES:
(6) DATATYPE: Integer, VALUES:
(7) DATATYPE: Nat_1, VALUES:
(8) DATATYPE: Nat, VALUES:
(9) DATATYPE: MODEL_TYPE, VALUES:
(10) DATATYPE: SIMTIME, VALUES:
(11) DATATYPE: DEGREE, VALUES:
(12) DATATYPE: RADIAN2, VALUES:
(13) DATATYPE: RADIAN1, VALUES:
(14) DATATYPE: KPH, VALUES:
(15) DATATYPE: KILOMETER, VALUES:
(16) DATATYPE: AF_MODELS, VALUES:

Will 'FUEL_LEVEL_TYPE' be derived from (subset of) one of the above types?
Choose which one or (0) to create new Base Type =>  0

Is this new type 'FUEL_LEVEL_TYPE' an enumerated type? => y

Enter one of the enumerated values or return when done => : EMPTY
Enter one of the enumerated values or return when done => : QUARTER
Enter one of the enumerated values or return when done => : HALF
Enter one of the enumerated values or return when done => : THREE_QUARTER
Enter one of the enumerated values or return when done => : FULL
Enter one of the enumerated values or return when done => :

Do you want to enter synonyms for 'FUEL_LEVEL_TYPE'? n

DATATYPE: FUEL_LEVEL_TYPE VALUES:  EMPTY QUARTER HALF THREE_QUARTER FULL

(0) None of these
(1) SELECT to use 'as is' in the Specification
(2) MODIFY it before SELECTing for the Specification
(3) DO NOT SELECT for use in the Specification

What do you want to do with this object? =>  1

OBJECT MODIFIED NEW DESCRIPTION IS:

ATTRIBUTE:  fuel_level : FUEL_LEVEL_TYPE
```

Figure 33 Screen Display: Creating a data type

4. The tool iterates through the list of enumerated values, placing each one entered into the type-values map.

5. A new *DomBaseType* or *DomDerType* is created and placed into the *has-obj-object* map.

6. The *Add-Object* database is returned to the *Perform-Add-Actions* function for further processing.

4.5.4 Adding New Objects to the Specification

After the backward chaining engine returns the database to the *Perform-Add-Actions* function, the *Add-Obj-To-Tree* function is called to find the proper place in the specification AST for the new item. If the new item is an operation, the function *Add-GOMT-op-To-Tree* is called to find the *GOMT-Class* that should store the new operation. This function tries to infer the proper class for the operation by mapping its predicates via the *map-ID-to-Object* function, described earlier, and getting the set of *GOMT-Classes* that hold the objects mapped to the predicate identifiers. If only one class is found, the operation is placed there. If no classes are found, it must be a pure operation and is placed into the *Op-Library* class. If more than one class is represented in the operation predicates, a complex while loop finds the lowest level aggregate class that includes all classes represented in the operation predicates. If all attempts to place the operation fail, the user is asked to choose the class.

New data types are simply added to the set of domain global types, and *GOMT-Classes* are added to the *has-primitive-classes* or *has-aggregate-classes* maps as appropriate. If a new item is the result of a modification action, the parent object that gets the new item is generally known and is identified in the *has-parent-objs* map of the *Add-Object* and can therefore be easily added to the AST below the parent object. If the new item created was a predicate by itself, which could happen when modifying the predicate of an operation, the predicate object is returned to the calling function and does not pass through the *Add-Obj-To Tree* function.

Except for predicates, the data dictionary is updated with the names of the objects in the subtree of the new item with a call to the *Add-to-Data-Dict* function. While in *Add-to-Data-Dict*, the user is asked if he wants to enter synonyms for the new objects, as shown in Figure 33. Finally, *Perform-Select-Actions* is called, which displays the new item on screen and asks the user if the new item should be selected for the specification or modified first, again shown in Figure 33.

4.6 Viewing the Specification

By choosing choice (7) from the main menu, the user can view the objects currently selected for the specification. The *Print-the-Spec* function is called to print the objects with the *eh-used* flag set to true. First the global data types are printed, then the global constants, followed by aggregate classes and primitive classes, and finally associations. The *Full-Selected-Obj-Description* function formats the object descriptions into

strings that are printed by the *Print-the-Spec* function. Figure 34 shows a printout of the specification after the example *CalcPropWt* function was created and added to the specification.

```
SELECTED PARTS FOR CURRENT SPECIFICATION

DATATYPE: Real
DATATYPE: FUEL_LEVEL_TYPE VALUES:  EMPTY QUARTER HALF THREE_QUARTER FULL

AGGREGATE CLASS: JetPropulsionSys  IS CONCRETE CLASS
  CONNECTION: fueltank IS EXACTLY ONE 'FuelTank'
  CONNECTION: jetengine IS EXACTLY ONE 'JetEngine'
  OPERATION:  CalcPropWt
    OUTPUT PARAMETER: prop_wt : Real
    PREDICATE:    prop_wt ! =fueltank.tank_weight + jetengine.engine_weight

PRIMITIVE CLASS: FuelTank  IS CONCRETE CLASS
  ATTRIBUTE:  fuel_level : FUEL_LEVEL_TYPE
  ATTRIBUTE:  tank_weight : Real

PRIMITIVE CLASS: JetEngine  IS CONCRETE CLASS
  ATTRIBUTE:  engine_weight : Real
```

Figure 34 Screen Display: A view of the selected specification in pretty print format

4.7 Saving the Specification

Choice (6) on the main menu takes the user to a sub menu with three options as shown in Figure 35. Choices (1) and (2) save the specification AST to a POB (Persistent Object Base) which can be loaded back into memory in a later REFINE session. The user is asked to enter a name for the POB file and the REFINE function *pob-dump-file* is called to perform the actual save.

```
(0) None of these
(1) Save 'In-Work' Specification to POB file
(2) Save Final (Cleaned-up) Specification to POB file
(3) Save Text Description of Specification to file

Choose the Save function you want to perform =>
```

Figure 35 Screen Display: The Save sub menu.

Choice (1) saves all unmodified objects from the original domain AST plus any new or modified items resulting from EH processing. This choice allows the user to save work in progress and return later to continue working on the specification where he left off. Before saving the file, the user is asked to give the specification a name, which is placed in the *spec-name* map of the *GOMT-DomainTheory* object. The next time the specification is loaded back into REFINE and passed to EH, the *Init-Spec-Tree* function checks the spec-name.

If the *spec-name* is defined, some initialization functions, such as deleting duplicate types, can be skipped because they were done on the initial run.

Choice (2) is used when the user is confident the application has been completely specified and wants to keep only the selected objects without the rest of the unselected domain objects. Before saving the specification to the POB file, a function called *Purge-Spec* deletes unused objects from the AST and removes the data dictionary subtree. This save should be chosen only after the clean-up functions have been run on the specification to ensure consistency. The specification AST resulting from the final save would then become the input to the design phase of development.

Choice (3) pretty prints the selected specification to a text file instead of to the screen.

4.8 Design Summary

This design chapter discussed the details of integrating the various parts of the EH tool. The AI techniques and algorithms used to make intelligent decisions were described, and several screen displays were shown to give the reader a feel for the user interface and the flow of the process. Chapter 5 discusses the functionality actually implemented, some problems encountered during implementation, and the methods used to evaluate the tool.

5 Implementation and Evaluation

This chapter describes the progress made during implementation and several issues that had to be dealt with. The method used for testing the EH and evaluation of its usefulness is also discussed.

5.1 EH Functionality Implemented

Not all EH requirements outlined on Chapter 3 were implemented in this version due to time constraints. Data types and operations have been particularly hard to define when working with the KBSE domain model because they contain predicates that can be very complex. For this reason, data types and operations received the most time and effort during implementation; and therefore, other types of domain items were implemented only partially or not at all. Figure 36 compares the EH requirements defined in Chapter 3 with the capabilities actually implemented in this version. A slash means the requirement has been partially implemented, for example, some parts of a class can be modified such as attributes and operations, but not states and events.

Section	Item Type	Select	Create	Modify	Delete
3.4.1.1	Primitive Classes	X		/	
3.4.1.2	Class Attributes	/		/	
3.4.1.3	Class Operations	X	X	X	/
3.4.1.4	States				
3.4.1.5	Events				
3.4.1.6	Transitions				
3.4.1.7	Parameters	X	X	X	/
3.4.1.8	Predicates	X	X	X	/
3.4.1.9	Data Types	X	X	X	
3.4.1.10	Constants	X	X	X	
3.4.1.11	Inheritance				
3.4.1.12	Associations	/		/	
3.4.1.13	Aggregate Classes	/		/	
3.4.1.14	Aggregate Operations	X	X	X	/

Figure 36 Capabilities implemented in this version of EH

The Class attributes receive only a partial selection rating on select and modify because the EH should be able to find and select the constraints that limit the value of the attribute. Association objects can be selected, but there is no capability to select the classes that are the connections of the association. The connections of associations and aggregate classes can be modified, but not all parts of an association class can be modified. The name of any object can be changed and the data dictionary will be updated; however the

capability to find and update all occurrences of the name in predicates is not complete. The capability exists to delete parameters, predicates, or entire operations; however, the tool does not do any safety checking to be sure the object being deleted is not used by other domain items.

The history database was not implemented. As mentioned in Chapter 3, a very simple history list could be implemented by creating an output text file to store descriptions of EH actions performed. The most likely place to perform the file writes would be in the functions *Perform-Select-Actions*, *Perform-Modify-Actions*, and *Perform-Add-Actions*, because those functions control changes made to the specification tree. A simple implementation like this would only provide the user with a chronological list of changes made to the AST. A more robust and useful history tool would allow the user to interactively view, undo, and redo specification actions, but was well beyond the scope of this research.

The clean-up functionality mentioned in Chapter 3 was partially implemented by the *Purge-Spec* function called by the *Save-Final-Spec* function. *Purge-Spec* calls the *preorder-transform* function to search the specification AST while applying the *Purge-Rule-Set* of forward reasoning rules. When a purge rule finds an object with the *eh-used* map set to false, it calls the *Remove-Object* function to erase the object and adjust the map that pointed to the erased object. If the map that contained the erased object was a set or sequence, the map to the erased object gets removed from the set or sequence. If the map only consists of a single object, the map is set to *undefined*. These functions take care of getting rid of the unnecessary objects, but other clean-up activities such as defining incomplete constant and data type declarations, and placing those constants and data types at the proper level were not implemented.

The capability to restrict certain actions on an object was not implemented. Marking an object as restricted against modification or deletion could done fairly easily by adding a map from *Obj-Objects* to a coded symbol, which would indicate if a restriction exists and if so, on what actions. A restriction map may alternatively be placed in the *Data-Elements* of the data dictionary. Implementing the restrictions in the EH would be much harder, because the restrictions would have to be checked in several places in the code. When a list of objects is displayed to the user, the restrictions would have to be checked and either printed out with the object description, or inhibit the objects from being displayed at all. More research needs to be done on this issue.

5.2 Maps Added to the Domain Model

Several maps were defined to help the EH perform its job. The map names and their purpose are described below.

- *has-aclass* – added as a non-tree attribute to *Parameter* objects to store a pointer to a *GOMT-Class* used as a data type of a parameter. E.g. `students: P Student`. The map *has-type-multiplicity* would also be used in this example to indicate *students* is a set. The *has-atype* map points to a *DomTypeObj* for those parameters declared as a data type.

- *has-DD* – added as the map from the root of the domain (*GOMT-DomainTheory*) to the root of the data dictionary (*Data-Dictionary*).

- *spec-name* – added to hold the name of the specification if the user decided to suspend the EH session and save the in-work specification until later.

- *eh-used* – added to all *Obj-objects* as a flag to indicate if the object has been selected to be included in the specification.

- *eh-pred-used* – added to all *Predicate* objects as a flag to indicate if the predicate has been selected to be included in the specification.

- *pred-mapped* – added to *Predicate* objects whose identifiers have been mapped to domain objects by the *Map-ID-to-Obj* function.

- *ID-wrd* – added to predicate *IDname* objects to aid the data dictionary in finding matching names in the domain.

- *id-obj* – added to predicate *IDname* object to point to the domain object it represents.

- *is-element-of* – added to predicate *IDname* objects to map the two identifiers of an '∈'expression, for example, `s ∈ students`.

- *is-BasicDecl-of* – added to predicate *IDname* objects to map the two identifiers of a declaration expression, for Example, `f: FuelTank`.

- *has-component-var* - added to predicate *IDname* objects to map the two identifiers of a component expression, for example, `FuelTank.fuel_level`.

5.3 Implementation Difficulties Encountered

This section describes some technical and coordination issues that arose during the prototyping phase of this research.

5.3.1 Parsing Predicates

The EH needed some method for the user to enter predicates such as invariants and especially pre-conditions and post-conditions. Since the user interface is text based, the options were to build some kind of parsing grammar or try to use something that already existed. The U-zed parser built by Wabiszewski [16] was already built and being used for parsing Z schemas from LaTeX files in a batch mode. The problem was how to get a text string typed in by the user during run time into a predicate tree that could be manipulated by the EH. There are two REFINE functions available for parsing, *parse-from-file* and *parse-from-string*. The parse-from-string function has three parameters, the string to be parsed, the grammar to be used, and a flag to indicate how errors are handled. Since the EH only needs to parse predicates, a subset of the existing grammar used for domain parsing was needed. The previous grammar files called *UtoolKit* and *Uzed* were modified to just handle predicate theory and renamed *PredToolKit* and *Upred*. These two new grammar files are compiled before the EH code files and so become the grammar that EH uses for parsing predicates. This approach may not be completely satisfying since the user must enter predicates in proper Z notation. It does provide a sound, structured way to implement and test the feasibility of this version of the EH.

5.3.2 Representing Function Calls in Z Predicates

Representing function calls in the transformation from the graphical functional model to the Z schema to the domain AST was the cause of some confusion. Consider the example in Figure 37. The Data Flow Diagram shows the output of *DetermineFuelWeight* going to the input of *CalcTotalWeight* and is reflected in the Z schemas. However, when parsed into the domain tree, there is an implicit requirement made that the name of the input parameter must match the name of the corresponding output parameter. Indeed, if the name of the output parameter of *DetermineFuelWeight* was changed, the *CalcTotalWeight* function would have no idea where its input would come from. A common way to implement these functions is shown in the REFINE code implementation of Figure 37. The *CalcTotalWeight* function uses a function call to the

DetermineFuelWeight to get the *fuel_weight* rather than expecting the value to be passed in as a parameter. The problem was the misunderstanding of how the U-Zed parser handled functions in predicates.

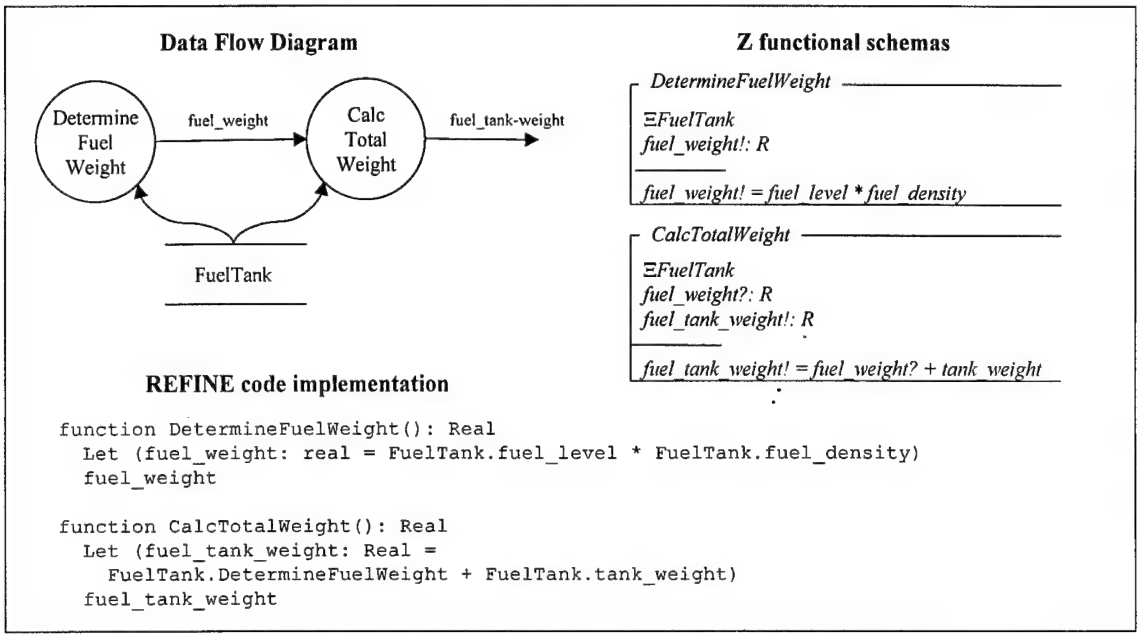

Figure 37 Inconsistency between Graphical and code representation

The Z notation allows for most of the arithmetic, relational, and logical expressions to be written using Infix notation, e.g. `X = Y + Z`. There are, of course, many standard and defined operations represented in Prefix notation such as `SquareRoot(x)`, or `power(x, n)`. There was some early confusion about if and how prefix operations would parse into the predicate tree. After some study and testing, it was discovered that the parser uses two slightly different notations concerning prefix operations. If there is only a single input parameter, simply put a space between the operation name and the parameter e.g. `Y = SquareRoot x`. If there are more than one input parameter, the parser parses them in as a single tuple entity using the familiar parentheses representation: `W = Power(x, n)`. Given either form of function call, the functions are parsed into a *FunctionApp-Expr* subtree in the *Unified-Object* model. Discovering this capability helped eliminate some uncertainty in the specification model. However, if a function has no input parameters, the parser has no way to distinguish between a function name and any other variable type. In this case the function name is placed into a var-name-expr subtree, so some ambiguity still exists in this issue.

5.3.3 Mapping Predicate Variables to Domain Objects

It became apparent during prototyping that a great deal of knowledge could be gleaned from predicates if the domain objects they represent could be accessed. Choosing the data type of a parameter to be modified is one reason why the object corresponding to the predicate identifiers would be needed. The EH could initially limit the choices of data types to those types and classes represented in the related predicates, assuming that the parameter will often be the same type as other variables in the predicate. If the proper data type is not found in the initial set, the entire list of data types are displayed for the user. Another situation where the objects of the predicate identifiers are needed is when an operation is selected for use in the specification. Because the predicate identifiers represent other objects such as attributes and operations, those represented items should also be selected for use in the specification. For example, the predicate in the *CalcTotalWeight* operation may be `fuel_tank_weight = DetermineFuelWeight + tank_weight`. The identifier *tank_weight* represents an attribute in the *FuelTank* class and should be selected for the specification. The identifier *DetermineFuelWeight* represents another operation, which in turn has a predicate `fuel_weight = fuel_level * fuel_density`. The operation *DetermineFuelWeight* and the attributes *fuel_level* and *fuel_density* should also be selected for the specification. Mapping identifiers to their corresponding domain objects was very important in implementing these capabilities.

The problem was that the predicates are implemented in the Unified Object model and the predicate variables were not mapped or associated to the domain objects they represent except by name. Therefore, in order to gather the data types represented by the predicates, some processing code had to be created that would search the data dictionary for perfect matches in order to find the related domain objects. The domain objects represented by the predicate variables could be attributes, constants, operations, parameters, classes, connections, or locally bound variables. Usually there will be only one perfect match, but bound variables would have no match, and attributes, constants, and parameters can possibly have more than one match if the name is used in different classes. For the case of bound variables, it is assumed there will be a declaration of the variable in the predicate, e.g. `f: FuelTank • f.fuel_level <= f.capacity`. Since the declaration parses in as a BasicDeclSeq expression in the predicate tree, the type or class of the bound variable can be found by looking at its declaration.

In the cases where more than one perfect match is found, the variable should be part of a component-expr, e.g. `f:FuelTank j:JetEngine • total_weight = f.weight+j.weight.` In this example, the attribute *weight* is defined in two classes, but each is prefixed with the bound variable indicating it is a component of a declared class. By getting the class associated with the bound variable, the attribute type can be found by looking to the matched attribute object that is in the class of the bound variable.

The function *Map-ID-to-Obj* sets a non-tree pointer from the IDname object of the predicate to the domain object it represents. It was discovered during prototyping that this function came in very handy during selection actions. When an operation is selected for use in the specification, the other domain objects represented by the predicate variables must also be selected for the specification. The *Map-ID-to-Obj* function finds those domain objects so their *eh-used* flags can be set.

5.3.4 Selection of Specification Items

When selecting an item with predicates, the objects represented by the predicate variables are also selected. These selected objects often have other objects in their subtree such as data types and other predicates. Since these selection actions turn out to be a recursive process, the *Select-Spec-Items* function is called recursively until all related objects are selected. Initially, it was assumed that the predicate objects would not need an *eh-used* flag to indicate selection, however, it is possible that the recursive selection functions could encounter the same predicate twice and end up in an infinite loop. Therefore, an *eh-pred-used* map was added to the predicate object and checked in the *Select-Spec-Items* function before processing the predicate variables to avoid infinite recursion.

5.3.5 Map from the New Object to the Parent Object

While developing the functionality for adding and modifying AST items, a problem arose regarding the information known about the action in progress. Sometimes when adding an item to the specification, very little is known about where the new item will be placed in the tree, but other times the parent objects, data types, or values may already be present in the AST. For example, when a new system output is identified and an operation must be created, it is not known to which class the new operation will belong. But, if the user is modifying a class and indicates that a new operation for the class is needed, then the EH should have the flexibility to use some information from the parent class while creating the new operation. The *has-parent-obj*

map was added to the *Add-Object* database to store the parent object if known. This map can be used when adding the new item to the specification and when looking for other domain information related to the new item.

5.3.6 Deleting Duplicate Types

During the initialization function, the *Delete-Duplicate-Types-Rule* calls *Delete-Duplicate-Types* function when two duplicate global data types are found in the domain. Duplicates can happen when two or more LaTeX files containing Z domain schemas with duplicate data type declarations are parsed into the domain tree. For every declared data type, a new *DomTypeObj* is created in the domain tree without checking for duplicates. When processing in the EH, duplicate data types can show up in object lists and cause confusion. It should have been fairly simple to look for duplicate global types by using the REFINE function called *term-equal?*. The *term-equal?* takes two arguments, which are object base trees, and returns true if its two arguments are isomorphic and all map values are equal. When *term-equal?* is true the *Delete-Duplicate-Types* function deletes one of the data type objects. The *term-equal?* function did not work as advertised in this case. It would not return true for duplicate objects; therefore, this functionality was disabled until a working solution could be found.

5.3.7 Problems with POB save

Saving the specification AST to a file as a POB (Persistent Object Base) was another problem area. The general way of saving a POB is to declare a dump-descriptor variable for each AST or subtree that should be saved. The dump-descriptor identifies a set of class names and a sequence of maps for each class that should be saved. Since this approach can get very tedious for large domains, REFINE provides a function called *make-dump-desc-for-class-tree*, which claims to automatically save all defined maps of all objects below the root node passed in to the function. This function is supposed to be a convenient short cut to actually naming every map in the domain the user wants to save. Two other optional arguments of *make-dump-desc-for-class-tree*, *atts-always-to-dump* and *atts-never-to-dump*, allow the user to specify AST objects as exceptions when saving the POB. This function did not work as advertised. It would only save AST objects if all the desired maps were actually named in the argument list. Three such dump-descriptors had to be defined because of the differing inheritance paths: one for *GOMT-DomainTheory*, one for *GOMT-Objects*, and one for *Unified-Objects*. These

three descriptors are then merged with a function called *merge-input-dump-descs* that returns a dump-descriptor containing the three merged descriptors. This merged descriptor is then used as the argument to the *pob-dump-file* function that writes the POB to a file.

5.4 Evaluation

Although it is generally claimed that automation of a manual process will improve productivity, the claim cannot be substantiated without performing some sort of metrics evaluation. Several metrics could be used to evaluate the performance of an automated EH. Possible methods include:

- Use several domains and written specifications and compare the averages of the total time taken to produce a specification AST using a manual process versus using the EH.
- Have several people try the tool and take a survey of their opinions on ease of use, timesavings, suggested improvements, etc.
- Observe a few people who are familiar with the AFIT KBSE system while they use EH and log the comments they make as a way to get feedback on the usefulness and ask their opinion as to whether they would prefer an EH tool over the previous manual method
- Make a smaller scale test by breaking down the manual process of creating a specification item into its subtasks. Perform several specification actions on various items while timing each of the subtasks using a manual method. Then, perform the same actions with the EH. Since the subtasks of the two methods are different, the manual tasks would need to be summed and compared to the total EH time.

The first two methods were not possible because there are currently very few detailed specifications, there are few people who are familiar enough with the AFIT KBSE system to get a good sampling, and the functionality of this EH version was not complete enough to produce entire specifications. The third method is good for soliciting feedback; however, opinions are subjective and hard to quantize into meaningful metrics. Therefore, the fourth method seemed like the most promising way to objectively evaluate the EH.

One problem with comparing the manual "old" way creating specifications from a domain to the "new" automated tool, was the lack of a manual process to begin with. There is fuzzy line between a domain tree and a specification tree. Ideally, the domain is more general and could be used to create several specifications for applications within that domain. Since there was not a well defined algorithm for building a specification from the domain, and since the AST structure is the same for both the domain and the

specification, in practice the software development process usually started by defining the specification requirements directly. Before meaningful metrics could be made, the manual process for refining a specification from a domain needed to be defined.

5.4.1 The Manual Process Defined

The manual process for defining a specification assumes a domain has been created and is fairly complete. The domain definitions generally begin as Z schemas written on paper as they are created. Once on paper, the domain engineer enters the Z schema information into a template (one GOMT-Class per template) that can be read by a LaTeX parser and understood by the U-Zed parser. After successful parsing, the LaTeX files are saved for future updates and the domain information resides in the domain AST where it can be saved to a POB file. The most likely specification process would start with the saved LaTeX files and the engineer should have some written requirements for his particular problem.

1. Locate the directory and LaTeX file that defines the class needed. This assumes the engineer is quite familiar with the domain since he must know the class needed to add new operations. He would probably need to refer to Z schema printouts and pictures of the domain object model.

2. Open the LaTeX file in a text editor, key in the definitions needed for the specification, and save the file under a new name. This requires the engineer to understand LaTeX syntax and the peculiarities of the U-Zed parser.

3. Open the printDD file; update with the name of the new specification file, and save it. The printDD file defines the title page and other configuration if the Z schemas are printed out. This step is not required if a graphical print out of the Z specification is not needed.

4. Run the new file through the LaTeX compiler. If the compiler fails because of errors, repeat steps two and four until successful. Repeat steps 1-4 for each file that needs to be updated for the specification.

5. Start up REFINE then start up Afittool. Choose to parse the new LaTeX file into the new domain, then iteratively append all other new specification files to the domain tree.

6. When all files have been appended, choose to save the domain to a POB file and give it a name; or begin the transformation to the design AST.

When finished with these steps, the specification should theoretically be ready for the design phase. However, some problems may occur due to the lack of consistency checks on the specification. Unless the engineer manually checks for and corrects them, the following problems will show up in the design AST.

- Predicate names are not checked for correct spelling or consistency with the names of the objects they represent. To overcome this problem, the engineer would probably need to print out the Z schemas of the domain for reference to the existing domain item names.

- Items such as class attributes or data types not used or needed in the application will not be purged from the specification AST. The engineer may not care if there are extra unnecessary objects in the design. But they could cause confusion downstream in the development when a designer decides to use one of those objects for another purpose since it is "there anyway". It could increase storage requirements and hinder performance if extra fields are kept in a database table with a million records. To avoid this problem, the engineer may need to perform a search through all the files for each domain object to see if it is required by a predicate somewhere, which would be extremely time consuming and error prone. The problem begs for some type of automated approach.

- Data types and constants declared in two or more class files will be duplicated in the specification AST. To avoid this duplication, the engineer would have to purposefully compare the class files or Z printouts and delete the duplicates from all but one file.

Checking these problems manually can add a great deal of time to the specification process and can be tedious, which adds to the risk of human error.

5.4.2 Standard Comparison Specifications

Since the EH was working for only a limited set of object types and actions, the specifications chosen as standards for comparison purposes had to be implementable using the EH. The Cruise Missile and the School were chosen as the two domains used as input when building the partial specifications described below.

For the Cruise Missile, perform the following refinements and selections:

1. Make all the weights a data type called KGS (kilograms), which is a real number >= zero
2. Make all fuel rates a data type called LITERS/SEC, which is a real number >= 0
3. Define a data type called FUEL_LEVEL_TYPE for fuel_level to be a real number >= 0 and <= 1.
4. Define a data type for capacity to be LITERS, which is a real number.

5. Create an output function to calculate the current fuel amount in Liters from (fuel_level * capacity).

6. Delete the predicate `fuel_level <= capacity` from the *FuelTank* class.

7. Change the predicate in the *DetermineFuelWeight* function to use a call to the function defined in the previous bullet (5) instead of the *fuel_level* variable.

8. Create a function that outputs the total weight of the CRUISE MISSILE in KGS calculated from the weights of its components then select this function for use in the specification.

9. Create and select an internal function that synchronizes the flow rates of the fuel tank and jet engine to the *actual_flow_rate* of the throttle

For the School, select the domain items required to:

1. Output the set of students advised given a faculty member.

2. Output the number of Master's degree students advised by a given faculty member.

3. Output the set of sections taught by a given faculty member.

4. Output the total number of students taught by a given faculty member.

Since the partial specifications above require Z predicates to be defined, it is assumed that the proper parseable predicates have already been formulated on paper, so neither approach is penalized for the time it takes trying to figure out the proper syntax.

5.4.3 Evaluation Results

The two sample specifications outlined in Section 5.4.2 were built using the six steps of the manual process described in Section 5.4.1 and the time taken for each step of the specification was logged. Next, the same two specifications were created using the EH tool, keeping track of the time taken for each step. After each specification process, the domain descriptions were printed out using an option provided by the AFITTOOL software. The descriptions list all the objects present in the specification AST. Appendix A first shows the objects selected during the EH version of the CRUISE MISSILE specification, followed by a list of additional objects in the manual specification that are not required. These additional objects remain in the specification because there is no way to purge unnecessary objects in the manual method unless the engineer removes them from the Z schemas in the LaTeX files. The SCHOOL specifications are also shown in

Appendix A. Since the dynamic model was not implemented in the EH, the states, events and transitions associated with the domain models were ignored during this evaluation.

This small test certainly does not have enough sampling points to perform any kind of statistical analysis, but several reasonable observations can be made from the results.

5.4.3.1 Time Comparison

Figure 38 summarizes the results of creating the manual version of the specifications versus the EH version using the same set of requirements. The speedup indicates how many times faster an enhanced method is versus the original method and is defined as Execution $time_{old}$ /Execution $time_{new}$. In both tests the EH method was a little more than twice as fast as the manual method, which shows some consistency.

	Cruise Missile	School
Manual Method	82	54
Elicitor-Harvester	40	25
Speedup	2.05	2.16

Figure 38 Time in minutes to complete specification process and the speedups obtained

This time test could be statistically validated given a large enough sampling space. Ideally, five or six people could each perform this time comparison on five or six different specifications. The results would be averaged with the lowest and highest single times thrown out. The times should form a normal distribution and a confidence interval could be calculated.

5.4.3.2 Correctness Comparison

During the manual process, the specification was not required to be purged of unselected objects, which resulted in a specification containing many unnecessary parts. Although these excess items may not immediately affect the specification, it is unnecessary overhead and could cause confusion downstream in the design phase. In Appendix A, the sections labeled as UNSPECIFIED MISSILE/SCHOOL COMPONENTS REMAINING IN MANUAL SPECIFICATION show several pages of excess parts of the domain that should be removed by the engineer to maintain correctness. Notice that there are also come duplicate data types defined in the list of *global data types*. These duplicates occur because they are defined in the Z schemas of more than one class and the U-Zed parser places all data types in the global area. Removing these excess items would be a tedious task that would add a lot of time to the manual process.

Checking for correctness is an area where automated tools generally excel. Just as a compiler checks the correctness of source code semantics, an EH could check the formal language semantics of domain predicates. Although extensive correctness checking capabilities have not been built into this EH version, some predicate checking was implemented as described in Sections 5.3.3 and 5.3.4. Predicate variables in newly created operations and data types as well as variables in predicates selected for the specification are compared to data dictionary elements to be sure of consistent spelling and to assure the objects represented by those variables are also selected for use in the specification. When the name of an object is modified, the EH updates the data dictionary and searches the specification tree to update predicate variables that map to the object being changed. In the manual process, it is quite possible that a predicate variable could be spelled differently than the object it represents and pass through the specification phase undetected - possibly causing confusion in the design phase.

Checking for errors and correctness are tasks generally performed much more efficiently with an EH than by a manual method. The benefit of automation grows as the size of the specification grows. For a small specification, a human may be able to manually check for errors and correctness, but as the number of classes in the specification grows, the number of details becomes overwhelming. As long as the error and correctness checking tasks can be described with an algorithm or a set of rules, an automated tool such as an EH will be able to perform the tasks much more quickly and accurately. Comparing the accuracy of the two methods would require extensive monitoring of the design process as well as verification and validation testing at the end of development for several applications. An evaluation would require an accurate count of the number of errors and inconsistencies encountered as a result of the specification process. Unfortunately, implementing such a test was beyond the scope of this thesis effort.

5.4.3.3 Ease of Use

Another way the EH demonstrates improvement is by simplifying the specification process. The manual method of creating a specification was loosely defined as a six-step process in Section 5.4.1. The first four steps require the engineer to find, open, modify, and save all LaTeX files containing Z schemas necessary for the specification. Step 5 indicates that the user needs to start up REFINE and AFITTOOL to parse the LaTeX files into the specification AST. The EH process starts at Step 5 except that the domain should be saved in a POB file and the engineer would choose option 12 from the AFITTOOL submenu, shown in Figure 39, and

enter the name of the domain POB file to load into the specification AST. The user would then back up to the main menu, also shown in Figure 39, and start up the EH by choosing option 2, which guides the user through the specification process.

```
------------------------------------------------------
     Welcome to the AFIT Software Transformation System
                       Version 0.4a
------------------------------------------------------

What would you like to do?
    0 - Exit AFITtool.
    1 - Perform Domain operations.
    2 - Perform Elicitor-Harvestor operations.
    3 - Perform Design operations.
    4 - Perform C++ Code Generation operations.
        ENTER YOUR CHOICE : 1

------------------------------------------------------
     Welcome to the AFIT Software Transformation System
                  ***>DOMAIN MODELER<***
------------------------------------------------------

What would you like to do?
    0 - Return to main menu.
    1 - Zstrip, parse, and create a domtree (2, 3, 4)
        2 - Zstrip a LaTeX domain file to zstrip.out.tex
        3 - Parse a Z-stripped file into a uzed AST.
        4 - Create a DOM AST from a uzed AST.
    5 - Parse a Z-stripped file and create a DOM (3 - 4).
        6 - Append to the DOM AST from a uzed AST.
    7 - Append to the DOM AST from Latex file (2,3,12).
    8 - Display the current domain model.
    9 - Output the domain model as .out file.
   10 - Save the domain model as a POB file.
   11 - Zap the current domain model.
   12 - Load the domain model from a POB file.
   13 - Display the current domain using grammar.
   14 - Output the domain using grammar as .lst file.
   15 - Output the domain using architecture language Acme.
        ENTER YOUR CHOICE :
```

Figure 39 The AFFITTOOL main menu and domain functions submenu.

Measuring the ease of use metric can be somewhat imprecise and was not performed in this research because there were not enough qualified people for a good statistical sampling. Assuming there were enough people for this test, the ease of use could be measured subjectively by asking each person to create a small specification with both the manual method and the EH, then ask them to fill out an opinion survey about their experience with both methods. The survey would ask them to rate the ease of use and intuitiveness of the two methods on a scale of one to five. The survey results would then be averaged out to come up with a single

rating number for each method and the two numbers could be compared to give an indication of which method was easier for the user.

5.5 Implementation Summary

The chapter described the parts of the EH that were successfully implemented. Several changes made to the domain model to support the EH were listed and difficulties encountered during implementation were discussed. Methods used to try to evaluate the usefulness and effectiveness of the tool were described as well as metrics that could have been used given a larger sample space. Chapter 6 has some concluding remarks and lists several ideas for future research.

6 Conclusions and Recommendations

The quest for a robust Elicitor-Harvester continues to be a challenge. Although previous work declared an EH tool as feasible and promising, many parts of the tool remain unproven and unimplemented. Previous thesis work of Wright [3] and Cochran [6] was limited to a small single domain and illustrated the concept of choosing reusable components from a specific domain. However, for a knowledge-based system to be useful, it must be generic enough to store and allow manipulation of most domain types. Because an EH must operate within the framework of a larger KBSE system, truly proving feasibility of a generic EH requires a well-defined domain theory, or *meta-model* that can store all types of domain knowledge and a variety of domains to allow for testing of various aspects of differing specifications. Wright and Cochran did not have the luxury of a pre-existing generic domain model as a framework to build upon and had to create their own, forcing them to restrict the scope of research. Karagias's work [2] was performed after the DOM meta-model had been partially defined and so was able to study the problem with well represented object-oriented domains. The implementation examples of the previous thesis work focused on building something like a pump, a queue, or a propulsion system. Generally, the user was prompted to input some description or requirement about the object to be built and the EH would supply a list of parts that met the requirement and asked the user to select the part desired. This approach works well for domains where some aggregate object is being built. However, many real world specifications require identification of operations acting upon the objects, states of the system, and the events that trigger operations and state changes. Identifying these actions are usually the most difficult part of defining a specification; but by identifying operations, the objects and attributes required can usually be determined by an intelligent EH without bothering the user for such details.

6.1 Conclusions

The scope of this research was not broad enough to explore states and events, but an in-depth study of operations was performed. This work uncovered many complex problems encountered while manipulating operations and especially predicates, but has also shown some promising results. Several benefits of this research are summarized below.

1. The simple data dictionary prototype, with its associated rules and functions, proved very useful in matching user input to existing domain items. A well-established data dictionary would allow a user to create a specification without the need to refer to hard-copy Z-schema definitions of the domain.

2. An inference engine was built in REFINE, which successfully performed backward chaining on a set of rules. Backward reasoning provided a good method for creating specification items such as operations. The recursive nature of the backward chaining algorithm allows for other objects needed for the specification, such as a new data type or another operation, to be created in the meantime while creating the original item. For example, while creating a new operation, it is discovered that the output parameter has a data type that doesn't exist in the domain. A *DomTypeObj* becomes the new goal and the backward chaining engine is called recursively to create the new data type then return to the operation creation process when complete.

3. Forward reasoning was successfully used to modify specification objects. The REFINE *transform* functions were used first to access rules to determine the modifiable attributes of an object, then to find the rules needed to guide the user through the modification process.

4. It was shown that many objects could automatically be marked as selected for the specification when the user selects a single operation. Mapping predicate variables to the domain objects they represent is the key to this task. If a predicate is deemed necessary for a specification, then all the objects represented in that predicate as well as their data types and the classes to which they belong are also selected for the specification.

5. The ability to save the in-work specification or the final specification to a POB file was successfully implemented. Saving the in-work specification was fairly simple. One just needs to save all objects currently in the specification. However, saving the final version was more difficult because the unneeded objects had to be removed from the specification AST. Completing this task required searching the tree for unmarked objects, accessing the parent object, setting the parent pointer to *undefined* for a one-to-one map or removing the object from the set or sequence for a one-to-many map, and finally erasing the object.

6. Although this prototype is not complete enough to specify entire applications, analyzing its implemented capabilities indicates several observable improvements over the previous manual method of creating specifications. Small preliminary tests show that specifications can be created faster, with better error checking, and without extraneous objects by using the EH tool instead of the manual method.

6.2 Future Recommendations

The EH process is still a rich area for research. Following are several recommendations for future study.

- A Graphical User Interface (GUI) would allow more information to be displayed on each screen than the current text-based user interface. It could allow a user to select multiple objects in cases where the same action needed to be performed on several objects, for example, to change the data type of several attributes. The object model could be displayed graphically and the parts of a class could be viewed with mouse clicks on the class object box. Modifications could be made inside the object model boxes, data dictionary elements could be viewed and updated and relationship cardinality could be changed. There are currently very good data modeling tools on the market that perform similar functions for database development such as Erwin by Logicworks. An EH GUI may have many features of a data modeling tool in addition to object-oriented functionality to handle operations, states and events.

- A big drawback of formal methods is the general lack of expertise within the computer industry. There is a steep learning curve and they are generally difficult to understand. The need exists for an interface that can get the formal mathematical models out of the face of users and allow them to interact with the tool in a more natural language. This problem is exhibited in the EH when the user is asked to input a predicate in proper Z syntax. Obviously the user is required to have some knowledge of Z and set notation. A natural language interpreter could accept a user's description of a predicate in natural language and translate it into Z or another formal language. Of course there would need to be some structure imposed on the user's input, since natural language is so ambiguous.

- The history database is an important part of the EH, but was not implemented in this version. Any interactive tool should provide the user with the ability to undo certain actions if a mistake was made or requirements changed. The history database functionality, described in Section 3.2.2.2, still needs to be studied.

- The data dictionary is a tool that could have a function outside of the EH process. In fact, the data dictionary would stay with the domain AST and be available for all EH sessions using the domain. It could provide the ability to parse descriptions, synonyms, and maybe abbreviations of GOMT-Objects during the domain-engineering phase as objects are created. The data dictionary could also be expanded to allow the tool to *learn* new synonyms automatically during EH processing when the user matches an input to a domain object. Many improvements are needed for the data dictionary matching techniques to make them more efficient and precise.

- The role of the Elicitor-harvester could be expanded to become more of a "domain editor". The domain engineer could use it as a domain building/maintenance tool. The domain could be built from scratch without the need for Z-schemas, LaTeX files or parsing software. Updates could be made to the domain via the domain editor, and it could still be used as a specification tool. Within the domain editor the knowledge base administrator could set the restrictions on domain objects to allow or disallow certain actions on them. The restriction functionality also needs to be studied.

- As shown in Figure 36 of Section 5.1, there is a significant amount of functionality that has not been designed and implemented. There are still many more challenges to be faced with the dynamic model – the states, events, and transitions, as well as associations, classes and inheritance.

6.3 Final Comments

As Knowledge-Based Software Engineering matures, the need for automated tools to perform specification refinement will increase. By creating specifications that are complete and correct and building transformation software that automatically creates source code from the specification, application software can be maintained at the specification level. This ability will have a profound impact on way software is maintained and upgraded. A commercial grade Elicitor-Harvester tool is probably still a few years away, but it has certainly been proven feasible. The possible benefits of automation at the specification level of software demand further study in this promising field.

Bibliography

1. Hartrum, Thomas C. *An Object Oriented Formal transformation System for Primitive Object Classes.* AFIT School of Engineering draft report, March 19, 1998.

2. Karagias, Timothy. "Elicitation of Formal Software Specifications from an Object-Oriented Domain Model", Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1996. AFIT/GCS/ENG/96D-14, AD-A320 698.

3. Wright, Charles A. "Implementing an Elicitor-Harvester for the Automatic Reuse of Software Components", Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1995. AFIT/GCS/ENG/95M-02.

4. Thomas C. Hartrum, Timothy Karagias, "Generation of Object-Oriented Formal Software Specifications", Proceedings of IEEE 1997 National Aerospace and Electronics Conference (NAECON 97), Nov 1997.

5. McDermott, J. "R1: A Rule-Based Configurer of Computer Systems." *Artificial Intelligence*, vol. 19 no.1, September, pp.39-88.

6. Cochran, Jerry D. "A Knowledge-Based Elicitor-Harvester: Automating the Selection of Object-Oriented Components for Reuse", Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1995. AFIT/GCS/ENG/95D-02, AD-A305 776.

7. Avelino Gonzalez, Douglas Dankel. *The Engineering of Knowledge-Based Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

8. Brooks, Fredrick P. "No Silver Bullet", Computer, 10-18 (Apr 1987).

9. James Rumbaugh, Michael Blaha, William Primerlani Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

10. David Pautler, Steven Woods, Alex Quilici. "Exploiting Domain-Specific Knowledge To Refine Simulation Specifications", Proceedings of the 12th IEEE International Conference on Automated Software Engineering. 117-124. November 1-5, 1997.

11. Winston, Patrick Henry. *Artificial Intelligence, Third Edition.* Addison-Wesley Publishing Company, 1992.

12. Michael Blaha, William Premerlani. "A Catalog of Object Model Transformations", Presented at 3rd Working Conference on Reverse Engineering, Monterey, California, November 1996.

13. Stephanie Cammarata, Darrell Shane, Prasad Ram. "IID: An Intelligent Information Dictionary for Managing Semantic Metadata", Technical Report No. R-3856-DARPA prepared for the Defense Advanced Reasearch Projects Agency by RAND, Santa Monica, California, 1991.

14. Stephen J. Andriole, Charlton A. Monsanto, Lee Scott Ehrhart. "Knowledge-Based User-computer Interface Design, Prototyping and Evaluation – The Design Pro Advisory System", Technical Report No. AFRL-IF-RS-TR-1998-142 prepared for Air Force Research Laboratory, Rome, NY, by Drexel University, Philadelphia, PA, 1998.

15. Winfried Grassmann, Jean-Paul Tremblay. *Logic and Discrete Mathematics, A Computer Science Perspective*, Prentice Hall, Upper Saddle River, New Jersey, 1996.

16. Wabiszewski, Kathleen M. *Unification of Larch and Z-Based Object Models to Support Algebraically-Based Design Refinement: The Z Perspective.* MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, Dec 1994. AFIT/GCS/ENG/94D-24, AD-A289 234.

17. Beem, Charles G. *Extending the Formal Object Transformation Process to Support Algebraically-Based Design Refinement: The Larch Perspective.* MS thesis, Air Force Institute of Technology, 1995. AFIT/GCS/ENG/95D-01, AD-A303 748

18. DeLoach, Scott A. Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specification. PhD thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, June 1996. AFIT/DS/ENG/96-05, AD-A310 608.

19. Brooks, Fredrick P. "No Silver Bullet", Computer, 10-18 (Apr 1987).

20. *REFINE User's Guide*, Reasoning Systems, Palo Alto, California.

21. Laurel, Brenda. *The Art of Human-Computer Interface Design*, Addison-Wesley Publishing Company, Inc., 1991.

Appendix A: Output Specifications from Tests

CRUISE MISSILE EH SPECIFICATION

```
=======================================================
                  DOMAIN: Missile
=======================================================
global data types
  Real
  LITERS
    SUPERTYPE: Real
  FUEL_LEVEL_TYPE
    SUPERTYPE: Real
  LITERPERSEC
    SUPERTYPE: Real
  KGS
    SUPERTYPE: Real

primitive concrete class FuelTank
attributes
    output_flow_rate: LITERPERSEC;
    fuel_level: FUEL_LEVEL_TYPE;
    capacity: LITERS;
    tank_weight: KGS;
    fuel_density: Real;
end attributes
  Operations:
    GetFuelAmount (IN ;OUT current_fuel_amount )
     Predicates:
      current_fuel_amount!=(#31<a MULTIPLICATION-EXPR>)
    DetermineFuelWeight (IN ;OUT fuel_weight )
     Predicates:
      fuel_weight!=(#32<a MULTIPLICATION-EXPR>)
    CalcTotalWeight (IN ;OUT fuel_tank_weight )
     Predicates:
      fuel_tank_weight!=DetermineFuelWeight + tank_weight

primitive concrete class JetEngine
attributes
    engine_weight: KGS;
    maximum_fuel_flow_rate: LITERPERSEC;
    thrust_factor: Real;
    current_fuel_flow_rate: LITERPERSEC;
    current_thrust: Real;
end attributes
Constraints:
    engine_weight>0
    maximum_fuel_flow_rate>0
    thrust_factor>0
    current_thrust>=0
    current_fuel_flow_rate>=0
    current_fuel_flow_rate<=maximum_fuel_flow_rate
    current_thrust=(#33<a MULTIPLICATION-EXPR>)

primitive concrete class Throttle
attributes
```

```
    position_index: Real;
    maximum_flow_rate: LITERPERSEC;
    actual_flow_rate: LITERPERSEC;
end attributes
Constraints:
    position_index>=0.0
    position_index<=1.0
    actual_flow_rate=(#34<a MULTIPLICATION-EXPR>)

aggregate concrete class JetPropulsionSys
attributes
    prop_weight: KGS;
end attributes
components:
    fueltank: FuelTank;
    throttle: Throttle;
    jetengine: JetEngine;
end components:
Constraints:
    prop_weight=(#35<a COMPONENT-EXPR>)  + (#36<a COMPONENT-EXPR>)
  States:
  Class Events:
  Operations:
    SynchronizeFlowRates (IN ;OUT )
     Predicates:
      Predicate - #37<a CONJUNCT-PRED>

aggregate concrete class Airframe
attributes
    airframe_weight: KGS;
end attributes
Constraints:
    airframe_weight>=0.0

aggregate concrete class Missile
components:
    propsys: JetPropulsionSys;
    airframe: Airframe;
end components:
Operations:
    CalcMissileWeight (IN ;OUT missile_weight )
     Predicates:
      missile_weight!=(#38<a COMPONENT-EXPR>)  + (#39<a COMPONENT-EXPR>)
----------------------------------------
```

UNSPECIFIED MISSILE COMPONENTS REMAINING IN MANUAL SPECIFICATION

```
Global data types
    Boolean
    Digit
    Char
    Integer
    Nat_1
    Nat
    LiterperSEC
    MODEL_TYPE
    LITERperSEC
    KGS
    AF_MODELS
    KILOMETER
    KPH
    RADIAN1
    RADIAN2
    DEGREE
    KGS
    KGS

primitive concrete class Throttle
Operations:
    InitThrottle (IN ;OUT )
     Predicates:
      position_index'=0.0

primitive concrete class JetEngine
  attributes
    manufacturer: SEQ_Char;
    model_num: MODEL_TYPE;
Operations:
    SetRate (IN flow_rate ;OUT )
     Predicates:
      current_fuel_flow_rate'=flow_rate?
      current_thrust'=(#178<a MULTIPLICATION-EXPR>)

primitive concrete class FuelTank
  attributes
    tank_sim_time: SIMTIME;
     input_flow_rate: LiterperSEC
  Operations:
    InitFuelTank (IN ;OUT )
     Predicates:
      tank_sim_time'=0
      input_flow_rate'=0
      output_flow_rate'=0
      fuel_level'=0
      capacity'=0
      tank_weight'=0
      fuel_density'=0
    PredictTankFullTime (IN ;OUT overflow_event_time )
     Predicates:
      overflow_event_time!=tank_sim_time + capacity - (#179<a DIVISION-
      EXPR>)
    CalculateNetFlow (IN ;OUT net_flow_rate )
```

```
      Predicates:
       net_flow_rate!=input_flow_rate - output_flow_rate
     PredictTankEmptyTime (IN ;OUT tank_empty_event_time )
      Predicates:
       tank_empty_event_time!=tank_sim_time + (#180<a DIVISION-EXPR>)

aggregate concrete class JetPropulsionSys
  attributes
    prop_fuel: Real;
  Constraints:
    prop_fuel=(#185<a COMPONENT-EXPR>)
    Predicate - #186<an IMPLICATION-PRED>
    Predicate - #187<an IMPLICATION-PRED>
    (#188<a COMPONENT-EXPR>) =(#189<a COMPONENT-EXPR>)
    (#190<a COMPONENT-EXPR>) =(#191<a COMPONENT-EXPR>)
Operations:
    LoadFuel (IN fuel_load ;OUT )
     Predicates:
      (#198<a COMPONENT-EXPR>) =fuel_load?

aggregate concrete class Airframe
  attributes
    airframe_simtime: SIMTIME;
    model: AF_MODELS;
    drag_coef: Real;
    turn_coef: Real;
    attached_weight: KGS;
    applied_thrust: Real;
    R_{EARTH}: KILOMETER;
    af_damage: Nat;
    X: KILOMETER;
    Y: KILOMETER;
    Z: KILOMETER;
    V_x: KPH;
    V_y: KPH;
    V_z: KPH;
    A_x: Real;
    A_y: Real;
    A_z: Real;
    theta: RADIAN2;
    phi: RADIAN1;
    theta_d: RADIAN2;
    phi_d: RADIAN1;
    lat_0: DEGREE;
    lon_0: DEGREE;
    speed: KPH;
    X_E: KILOMETER;
    Y_E: KILOMETER;
    Z_E: KILOMETER;
    R_E: KILOMETER;
  components:
    tail_num: [ALPHANUM];
  Constraints:
    R_{EARTH}=6378.137
    drag_coef>=0.0
    turn_coef>=0.0
    attached_weight>=0.0
```

```
  applied_thrust>=0.0
      (#200<a MULTIPLICATION-EXPR>) =(#201<a MULTIPLICATION-EXPR>)  +
  (#202<a MULTIPLICATION-EXPR>)  + (#203<a MULTIPLICATION-EXPR>)
      X_E=(#204<a MULTIPLICATION-EXPR>)  - (#205<a MULTIPLICATION-EXPR>)
  + (#206<a MULTIPLICATION-EXPR>)  + (#207<a MULTIPLICATION-EXPR>)
      Y_E=(#208<a MULTIPLICATION-EXPR>)  - (#209<a MULTIPLICATION-EXPR>)
  + (#210<a MULTIPLICATION-EXPR>)  + (#211<a MULTIPLICATION-EXPR>)
      Z_E=(#212<a MULTIPLICATION-EXPR>)  + (#213<a MULTIPLICATION-EXPR>)
  + (#214<a MULTIPLICATION-EXPR>)
      (#215<a FUNCTIONAPP-EXPR>) =(#216<a FUNCTIONAPP-EXPR>)  + (#217<a
  FUNCTIONAPP-EXPR>)  + (#218<a FUNCTIONAPP-EXPR>)
Operations:
  InitAirframe (IN ;OUT )
   Predicates:
    af_damage'=0
    airframe_simtime'=0.0
    tail_num'=ABC123
    model'=V99
    airframe_weight=0.0
    drag_coef'=0.0
    turn_coef'=0.0
    attached_weight'=0.0
    applied_thrust'=0.0
    X'=0.0
    Y'=0.0
    Z'=0.0
    V_x'=0.0
    V_y'=0.0
    V_z'=0.0
    A_x'=0.0
    A_y'=0.0
    A_z'=0.0
    theta'=0.0
    phi'=0.0
    theta_d'=0.0
    phi_d'=0.0
```

SCHOOL EH SPECIFICATION

```
=====================================================
                DOMAIN: School
=====================================================
global data types
    Nat

primitive concrete class Faculty

primitive concrete class Student

primitive concrete class Section

primitive concrete class GradClass
  attributes
    program: PROGTYPE;
end attributes

aggregate concrete class WorkloadSystem
components:
    stu: {Student};
end components:
associations:
    assigned: stu (Student) (0..n)  <-->  (0..n) sect (Section);
    member_of: stu (Student) (0..n)  <-->  (1..1) grad (GradClass);
    r_advises: stu (Student) (0..n)  <-->  (0..n) fac (Faculty);
    teaching: fac (Faculty) (0..n)  <-->  (0..n) sect (Section);
end associations:
Operations:
  GetNumberStudentsTaught (IN faculty ;OUT num_students_taught )
    Predicates:
      num_students_taught!=(#41<a CARDINALITY-EXPR>)
  GetSectionsTaught (IN faculty ;OUT sections_taught )
    Predicates:
      sections_taught!=(#42<a SET-COMP-EXPR>)
  GetNumberMSStudents (IN ;OUT number_ms_students )
    Predicates:
      number_ms_students!=(#43<a CARDINALITY-EXPR>)
  GetStudentsAdvised (IN faculty ;OUT students_advised )
    Predicates:
      students_advised!=(#44<a SET-COMP-EXPR>)
-----------------------------------------
```

UNSPECIFIED SCHOOL COMPONENTS REMAINING IN MANUAL SPECIFICATION

```
 global data types
    Real
    Boolean
    Digit
    Char
    Integer
    Nat_1
    PERNAMES
    SSAN
    GENDER
    DATE
    ACADEMIC_RANK = (Instr Asst Assoc Prof)
    MONTH
    YEAR
    DATE
    PROGTYPE
    PROGTYPE = (GCS GCE GE GSS DS)
    SEQ_Char
      SUPERTYPE: Char
      CONSTRAINTS:     None.
      Has sequence multiplicity
    SEQ_Digit
      SUPERTYPE: Digit
      CONSTRAINTS:     None.
      Has sequence multiplicity
    CTYPE
    CNUM
    ALPHANUM

primitive concrete class Person
attributes
    lastname: PERNAMES;
    initial: Char;
    firstname: PERNAMES;
    birthdate: DATE;
    ssan: SSAN;
    sex: GENDER;
    height: Nat_1;
    weight: Nat_1;
    age: Nat_1;
end attributes
Constraints:
    age=(#23<a FUNCTIONAPP-EXPR>)

primitive concrete class Faculty subclass of Person
  attributes
    academic_rank: ACADEMIC_RANK;

primitive concrete class Student subclass of Person
  attributes
    gpa: Real;
  Constraints:
    gpa>=0.0
    gpa<=4.0
    age=(#24<a FUNCTIONAPP-EXPR>)
```

```
primitive concrete class GradClass
  attributes
    year: YEAR;
    month: MONTH;
    graddate: DATE;
    designator: SEQ_Char;

primitive concrete class Section
  attributes
    number: SEQ_Digit;
  Constraints:
    (#25<a CARDINALITY-EXPR>) =2

aggregate concrete class WorkloadSystem
  components:
    fac: {Faculty};
    sect: {Section};
    grad: {GradClass};
    curr: {Course};
    offer: {Offering};
    quarter: {Quarter};
    taught_as: sect (Section) (0..n)  <-->  (1..1) offer (Offering);
    offered: curr (Course) (0..n)  <-->  (0..n) quarter (Quarter) -->
    Offering;
```

Appendix B: Sample Domains

Z-Schema Definitions Used for

CRUISE MISSILE DOMAIN

Missile System Structure Definition

Missile

$propsys : JetPropulsionSys$
$airframe : Airframe$

Jet Propulsion System Structure Definition

JetPropulsionSys

$fueltank : FuelTank$
$throttle : Throttle$
$jetengine : \text{seq } JetEngine$
$prop\_weight : R$
$prop\_fuel : R$

$prop\_weight = fueltank.CalcTotalWeight + jetengine.engine\_weight$
$prop\_fuel = fueltank.fuel\_level$
$(fueltank.fuel\_level = 0 \Rightarrow throttle.maximum\_flow\_rate = 0)$
$(fueltank.fuel\_level > 0 \Rightarrow throttle.maximum\_flow\_rate = jetengine.maximum\_fuel\_flow\_rate)$
$fueltank.output\_flow\_rate = throttle.actual\_flow\_rate$
$throttle.actual\_flow\_rate = jetengine.input\_flow\_rate$

Jet Propulsion Functional Definition

LoadFuel

$\Delta JetPropulsionSys$
$fuel\_load? : R$

$fueltank'.fuel\_level = fuel\_load?$

Throttle Structure Definition

Throttle

$position\_index : R$
$maximum\_flow\_rate : R$
$actual\_flow\_rate : R$

$position\_index \geq 0$
$position\_index \leq 1.0$
$actual\_flow\_rate = position\_index * maximum\_flow\_rate$

Airframe Structure Definition

[*AF_MODELS, KILOMETER, KPH, RADIAN1, RADIAN2, DEGREE*]

Airframe

airframe_simtime : *SIMTIME*
tail_num : seq *ALPHANUM*
model : *AF_MODELS*
airframe_weight : *R*
drag_coef : *R*
turn_coef : *R*
attached_weight : *R*
applied_thrust : *R*
R_{EARTH} : *KILOMETER*
af_damage : *N*
X : *KILOMETER*
Y : *KILOMETER*
Z : *KILOMETER*
V_x : *KPH*
V_y : *KPH*
V_z : *KPH*
A_x : *R*
A_y : *R*
A_z : *R*
theta : *RADIAN2*
phi : *RADIAN1*
$theta_d$: *RADIAN2*
phi_d : *RADIAN1*
lat_0 : *DEGREE*
lon_0 : *DEGREE*
speed : *KPH*
X_E : *KILOMETER*
Y_E : *KILOMETER*
Z_E : *KILOMETER*
R_E : *KILOMETER*

$R_{EARTH} = 6378.137$
$airframe\_weight \geq 0.0$
$drag\_coef \geq 0.0$
$turn\_coef \geq 0.0$
$attached\_weight \geq 0.0$
$applied\_thrust \geq 0.0$
$(speed * speed) = (V_x * V_x + Vy * V_y + V_z * V_z)$
$X_E = -1 * sin\ lon_0 * X - sin\ lat_0 * Y + cos\ lon_0 * cos\ lat_0 * Z + R_{EARTH} * cos\ lon_0 * cos\ lat_0$
$Y_E = cos\ lon_0 * X - sin\ lon_0 * sin\ lat_0 * Y + sin\ lon_0 * cos\ lat_0 * Z + R_{EARTH} * sin\ lon_0 * cos\ lat_0$
$Z_E = cos\ lat_0 * Y + sin\ lat_0 * Z + R_{EARTH} * sin\ lat_0$
$square\ R_E = square\ X_E + square\ Y_E + square\ Z_E$

Fuel Tank Structure Definition

[*SIMTIME*]

FuelTank

tank_sim_time : *SIMTIME*
input_flow_rate : *R*
output_flow_rate : *R*
fuel_level : seq *R*
capacity : *R*
tank_weight : *R*
fuel_density : *R*

---

fuel_level ≤ *capacity*

FuelTank Functional Definitions

Determine Interval

Ξ *FuelTank*
Ξ *SimClock*
interval! : *SIMTIME*

---

interval! = *sim_time* - *tank_sim_time*

PredictTankFullTime

Ξ *FuelTank*
overflow_event_time! : *SIMTIME*

---

overflow_event_time! = *tank_sim_time* + *capacity* - *fuel_level* div *input_flow_rate*

PredictTankEmptyTime

Ξ *FuelTank*
tank_empty_event_time! : *SIMTIME*

---

tank_empty _event_time! = *tank_sim_time* + *fuel_level* div *output_flow_rate*

CalculateNetFlow

Ξ *FuelTank*
net_flow_rate! : *R*

---

net_flow_rate! = *input_flow_rate* - *output_flow_rate*

CalculateNewLevel

Δ *FuelTank*
Ξ *SimClock*
net_flow_rate? : *R*
interval ? : *SIMTIME*

---

fuel_level' = *fuel_level* + *interval*? * *net_flow_rate*?
tank_sim_time' = *sim_time*

DetermineFuelWeight

Ξ *FuelTank*
fuel_weight! : *R*

---

fuel_weight! = *fuel_level* * *fuel_density*

CalcTotalWeight

Ξ *FuelTank*
fuel_tank_weight! : *R*

---

fuel_tank_weight! = *DetermineFuelWeight* + *tank_weight*

SetInflow

Δ *FuelTank*
Ξ *SimClock*
flow_rate? : *R*

---

input_flow_rate' = *flow_rate*?
tank_sim_time' = *sim_time*

SetOutflow

Δ *FuelTank*
Ξ *SimClock*
flow_rate? : *R*

---

output_flow_rate' = *flow_rate*?
tank_sim_time' = *sim_time*

Jet Engine Structure Definition

[*MODEL_TYPE*]

JetEngine

manufacturer : seq *CHAR*
model_num : *MODEL_TYPE*
engine_weight : *R*
maximum_fuel_flow_rate : *R*
thrust_factor : *R*
current_fuel_flow_rate : *R*
current_thrust : *R*

---

engine_weight > 0
maximum_fuel_flow_rate > 0
thrust_factor > 0
current_thrust ≥ 0
current_fuel_flow_rate ≥ 0
current_fuel_flow_rate ≤ *maximum_fuel_flow_rate*
current_thrust = *thrust_factor* * *current_fuel_flow_rate*

Jet Engine Functional Definition

SetRate

Δ *JetEngine*
flow_rate? : *R*

---

current_fuel_flow_rate' = *flow_rate*?
current_thrust' = *thrust_factor* * *current_fuel_flow_rate'*

Z-Schemas Definitions Used for

SCHOOL DOMAIN

WorkLoad System Structure Definition

WorkLoadSystem

$fac : P\ Faculty$
$stu : P\ Student$
$sect : P\ Section$
$grad : P\ GradClass$
$curr : P\ Course$
$offer : P\ Offering$
$quarter : P\ Quarter$
$assigned : (stu \leftrightarrow sect)$
$member\_of : (stu \rightarrow grad)$
$r\_advises : (stu \leftrightarrow fac)$
$taught\_as : (sect \rightarrow offer)$
$teaching : (fac \leftrightarrow sect)$
$offered : (curr \times \text{quarter} \rightarrow Offering)$

Person Structure Definition

[*PERNAMES, SSAN, GENDER, DATE*]

Person

$lastname : PERNAMES$
$initial : CHAR$
$firstname : PERNAMES$
$birthdate : DATE$
$ssan : SSAN$
$sex : GENDER$
$height : N_1$
$weight : N_1$
$age : N_1$

$age = yearinterval(birthdate, TODAY)$

Faculty Structure Definition

$ACADEMIC\_RANK ::= Instr \mid Asst \mid Assoc \mid Prof$

Faculty

$academic\_rank : ACADEMIC\_RANK$

Person

Student Structure Definition

Student

$gpa : R$

Person

$gpa \geq 0.000$
$gpa \leq 4.000$
$age = yearinterval(birthdate, TODAY)$

Offering Structure Definition

[*OFFERING_CODE*]

Offering

$code : OFFERING\_CODE$

Section Structure Definition

Section

$number : \text{seq } DIGIT$

$\#number = 2$

GradClass Structure Definition

[*MONTH, YEAR, DATE. PROGTYPE*]

$PROGTYPE ::= GCS \mid GCE \mid GE \mid GSS \mid DS$

GradClass

$program : PROGTYPE$
$year : YEAR$
$month : MONTH$
$graddate : DATE$
$designator : \text{seq } CHAR$

Quarter Structure Definition

[*QYEAR, DATE, QNAME*]

QYEAR : seq *DIGIT*

$\#QYEAR = 2$

$QNAME ::= SU \mid FA \mid WI \mid SP \mid SS \mid FS$

Quarter

qname : *QNAME*
qyear : *QYEAR*
start : *DATE*
end : *DATE*

$start < end$

Course Structure Definition

[*CTYPE, CNUM, ALPHANUM*]

CNUM : seq *DIGIT*

$\#CNUM = 3$

Course

ctype : *CTYPE*
cnum : *CNUM*
ctitle : seq *ALPHANUM*
cdesc : seq *ALPHANUM*
creditHours : *N*
lectureHours : *N*
labHours : *N*
abetDes : *N*
abetSci : *N*
abetMath : *N*
abetOther : *N*

$creditHours = abetDes + abetSci + abetMath + abetOther$
$creditHours = lectureHours + labHours / 3$

Appendix C: Compilation Configuration

The EH system can be compiled using the following order:

```
(require-system "DIALECT")
(require-system "intervista")
(require-system "workbench")
(compile-and-load-lisp-file "lisp-utilities")
(compile-and-load-file "read-utilities")
; The next group supports the domain modeling effort.
(compile-and-load-file "uzed-dm2")
(compile-and-load-file "uzed-gram2")
(compile-and-load-file "utoolkit-dm")
(compile-and-load-file "utoolkit-gram")
(compile-and-load-file "utility")
(compile-and-load-file "zstrip")
; The next group supports both domain modeling and specification (EH).
(compile-and-load-file "domain")
(compile-and-load-file "analyze")
(compile-and-load-file "domlist")
(compile-and-load-file "uzed2dom")
(compile-and-load-file "dom-grammar")
(compile-and-load-file "domsave")
(compile-and-load-file "dummyfcts")

; These following files are not needed for EH but you may get an error
;when compiling AFITTOOL if they are not available

; The following is for use with domain model counter3.tex only.
; (compile-and-load-file "testcnt3")
; (compile-and-load-file "domtool")
; The above are sufficient to populate the domain tree.
;     Can be invoked from menu by also including afittool (below).
;-----------------------------------------------------------------
; (compile-and-load-file "ehtool")
; The above are sufficient to generate domain specifications.
;     Can be invoked from menu by also including afittool (below).
;-----------------------------------------------------------------
; (compile-and-load-file "xforms")
; (compile-and-load-file "dom-refine")
; (compile-and-load-file "destool")
; The above are sufficient to transform specs to REFINE designs.
;     Can be invoked from menu by also including afittool (below).
;-----------------------------------------------------------------
;(load "~hartrum/kbse/code/d2c")

(compile-and-load-file "afittool")

;The following files are needed to run the EH
(compile-and-load-file "eh_pred_gram")
(compile-and-load-file "eh_predtoolkit_gram")
(compile-and-load-file "eh_dummyfcts")
(compile-and-load-file "eh_DD")
(compile-and-load-file "eh_domain")
(compile-and-load-file "eh_domsave")
(compile-and-load-file "eh_functions")
```

```
(compile-and-load-file "eh_fact_base")
(compile-and-load-file "eh_rule_base")
(compile-and-load-file "eh_back_eng")
(compile-and-load-file "eh")
```

File descriptions:
eh_pred_gram.re – Pulled the predicate part of the grammer from the uzed-gram2.re file and made a couple of other changes to accommodate reading predicate strings from the user during runtime.

eh_predtoolkit_gram.re – Copied from the utoolkit-gram.re file but with some minor changes to accommodate the eh_pred grammer.

eh_dummyfcts.re – function headers definitions to allow compilation. Cures the circular referencing of functions between two files.

eh_domain.re – The EH definitions added to the domain AST created by the domain.re file

eh_domsave.re – This is the latest version of the domsave.re utility for saving POB files

eh_functions.re – Several supporting functions for the EH

eh_fact_base.re – The structures used as the fact bases or databases

eh_rule_base.re – The sets of rules used during reasoning when adding and modifying objects.

eh_back_eng.re – The backward chaining engine that controls the manipulation of Add rules.

eh.re – The main functions used for the EH.

The Elicitor-Harvester can be started from the AFITTOOL menu.
First start AFITTOOL from the REFINE prompt with

```
.> (runtool)
```

Load a domain model by choosing the "Perform Domain Operations" option from the AFITTOOL main menu, then either load a domain from a saved POB file or create a domain by parsing in LaTeX files by choosing the corresponding option.

After the domain is loaded, back up to the main menu and select the option to "Perform Elicitor-Harvester Operations".

Vita

Captain Gary L. Anderson was born on February 16, 1962 in Rupert, Idaho. He attended High School in San Leandro, California and graduated from San Leandro High School in 1980. He enlisted in the Air Force in 1986 and worked as a calibration technician in a Precision Measurement Equipment Laboratory (PMEL). After attaining the rank of Staff Sergeant in three and one-half years, he was selected for the Airman Education and Commissioning Program (AECP) in 1990. He was reassigned to Wright State University in Dayton, Ohio where he received a Bachelor of Science degree in Computer Science in June 1993. After graduating as Suma Cum Laude, he attended Officer Training School (OTS) and was a member of the class that moved OTS from San Antonio, Texas to Maxwell AFB in Montgomery, Alabama. His first Officer assignment was in the Air Force Pentagon Communications Agency at the Pentagon in Washington, DC, where developed and maintained database systems in support of the DOD Presidential budget. In 1997, he again moved to Dayton, Ohio to attend the Air Force Institute of Technology.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE

AN INTERACTIVE TOOL FOR REFINING SOFTWARE SPECIFICATIONS FROM A FORMAL DOMAIN MODEL

5. FUNDING NUMBERS

6. AUTHOR(S)

Gary L. Anderson, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
2950 P Street
Wright-Patterson AFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/99M-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Roy F. Stratton
AFRL/IFTD
525 Brooks Rd.
Rome NY 13441-4505
(315) 330-3004

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Thomas C. Hartrum, Ph.D.
(937) 255-3636 X4581
Thomas.Hartrum@afit.af.mil

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This work examines the process for refining a software specification from a formal object-oriented domain model. This process was implemented with interactive software to demonstrate the feasibility and benefits of automating what has been a tedious and often error-prone manual task.

The refinement process operates within the framework of a larger Knowledge-Based Software Engineering system. A generic object-oriented representation is used to store a domain model, which allows the specification tool to access, select, and manipulate the required objects to form a customized specification. The specification is also stored as an object-oriented model, which in turn can be accessed by a design tool to transform the specification into source code.

The tool has been designed as an interactive program that helps guide the user through the process of building the specification. The tool has been named the Elicitor-Harvester because of the functions it performs. It elicits application requirements from the user and harvests pre-existing knowledge from the formal domain.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Specification, Refinement, Formal Method, Knowledge Base, Domain Model, Domain Editor, KBSE, Elicitation, Rule Base, Software Engineering.	124
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL